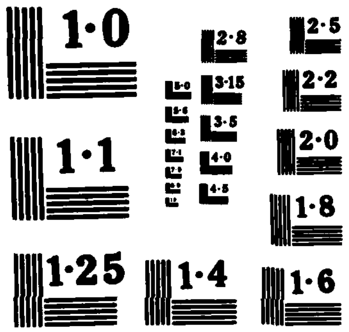
AD-A158 011 THE LOGLISP PROGRAMMING SYSTEM(U) SYRACUSE UNIV NY 1/2
J A ROBINSON ET AL. MAY 85 RADC-TR-85-89
F30602-81-C-0024

UNCLASSIFIED F/G 9/2 NL


1·0
2·8
2·5
3·15
2·2
3·5
2·0
1·1
4·0
4·5
1·8
1·25
1·4
1·6

2

AD-A158 011

RADC-TR-85-89
Final Technical Report
May 1985

THE LOGLISP PROGRAMMING SYSTEM

Syracuse University

J. A. Robinson, E. E. Sibert and K. J. Greene

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

85 8 14 036

RADC-TR-85-89 has been reviewed and is approved for publication.

APPROVED: Philip B. Tarbell III

PHILIP B. TARBELL III, Captain, USAF
Project Engineer

APPROVED: [illegible]

RAYMOND P. URTZ, JR.
Technical Director
Command and Control Division

FOR THE COMMANDER: [illegible]

RICHARD W. POULIOT
Plans Office

PREFACE

LOGLISP is basically ZETALISP [Moon, Stallman and Weinreb 1983] with a logic programming system embedded within it. The LOGLISP user we have in mind is thus (ideally) someone who is familiar with ZETALISP (or, at least, LISP); we do not in this manual address the non-LISP community, beyond some general discussion in the first few chapters.

Nor do we assume the user is already familiar with logic programming or the earlier background involving resolution theorem-proving. The early chapters attempt to provide an overall view of the essential ideas in a fairly general setting. In particular no prior acquaintance with PROLOG is assumed. In order to distinguish our work from that of our PROLOG colleagues (which, and whom, we esteem highly) the logic programming system within LOGLISP is called LOGIC. Thus we have: LOGLISP = LOGIC + LISP. The present version of LOGLISP has been improved considerably over earlier versions, both in the efficiency of the implementation and in the incorporation of several new features which we believe will be found useful.

LOGIC differs in a number of ways from the well-known PROLOG implementations of logic programming [Roussel 1975], [Warren 1977], [Roberts 1977], [Clark 1979]. The most noteworthy difference is that LOGIC is simply a set of new LISP primitives designed to be used *freely* within LISP programs. These primitives are invoked in the ordinary LISP manner by function calls from the terminal or from within other LISP programs. They return their results as LISP data objects which can be subjected to analysis and manipulation. Each of the logical procedures comprising a LOGIC knowledge base is a LISP data object stored (like the definition of an ordinary LISP procedure) among the information concerning the identifier which is its name.

Thus one calls LOGIC from within LISP. It is also possible to call LISP from within LOGIC. The identifiers used as logical predicate symbols, function symbols and individual constants within a knowledge base or query can be given a LISP meaning by the ordinary LISP method of definition or assignment. Some identifiers (CAR, CONS, PLUS, etc.) already have a LISP meaning imposed by the system. Thus every logic construct (term, or atomic sentence) is capable of being interpreted as a LISP construct. During the "deduction cycle" of LOGIC each logic construct is "evaluated" as a LISP construct, according to a suitably generalized notion of evaluation, called

"LISP-reduction".

The effect of this LISP-reduction step within each deduction step is to make available to the LOGIC programmer virtually the full power of LISP. This makes trivially easy the "building-in" of "immediately evaluable" notions - but far more than that. In particular, LOGIC calls can be made from within LOGIC calls.

The design and implementation of LOGIC was partially supported by the Rome Air Development Center of the United States Air Force under contracts F30602-77-C-0121 and F30602-81-C-0024, by the National Science Foundation under grant MCS-77-20780, and by the University of Edinburgh under a grant from the Science Research Council of the United Kingdom.

TABLE OF CONTENTS

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S): RADC-TR-85-89
6a. NAME OF PERFORMING ORGANIZATION: Syracuse University; 6b. OFFICE SYMBOL (If applicable):	7a. NAME OF MONITORING ORGANIZATION: Rome Air Development Center (COES)
6c. ADDRESS (City, State and ZIP Code): 313 Link Hall, Syracuse NY 13210	7b. ADDRESS (City, State and ZIP Code): Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Rome Air Development Center; 8b. OFFICE SYMBOL (If applicable): (COES)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F30602-81-C-0024
8c. ADDRESS (City, State and ZIP Code): Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
62702F	5581	19	14

11. TITLE (Include Security Classification): THE LOGLISP PROGRAMMING SYSTEM

12. PERSONAL AUTHOR(S): J. A. Robinson, E. E. Sibert, K. J. Greene

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM Feb 81 TO Feb 84	May 1985	134

16. SUPPLEMENTARY NOTATION: N/A

17. COSATI CODES

FIELD	GROUP	SUB. GR.
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Loglisp; Logic Programming; Lisp; Programming Languages; Programming Environments

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

LOGLISP is basically ZetaLisp with a logic programming system, LOGIC, embedded within it. LOGIC differs in a number of ways from the well-known PROLOG implementations of logic programming. The most noteworthy difference is that LOGIC is simply a set of new LISP primitives designed to be used freely within LISP programs. These primitives are invoked in the ordinary LISP manner by function calls from the terminal or from within other LISP programs. They return their results as LISP data objects which can be subjected to analysis and manipulation. Each of the logical procedures comprising a LOGIC knowledge base is a LISP data object stored (like the definition of an ordinary LISP procedure) among the information concerning the identifier which is its name. Thus, one calls LOGIC from within LISP.

It is also possible to call LISP from within LOGIC. The identifiers used as logical predicate symbols, function symbols, and individual constants within a knowledge base or

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Philip B. Tarbell, Captain, USAF	22b. TELEPHONE NUMBER (Include Area Code): (315) 330-3564; 22c. OFFICE SYMBOL: RADC (COES)

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE. UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

query can be given a LISP meaning by the ordinary LISP method of definition or assignment. Thus, every logic construct is capable of being interpreted as a LISP construct. During the "deduction cycle" of LOGIC each logic construct is "evaluated" as a LISP construct, according to a suitably generalized notion of evaluation, called "LISP-reduction". The effect of this LISP-reduction step within each deduction step is to make available to the LOGIC programmer virtually the full power of LISP. Additional keywords

programming languages

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

CHAPTER 1

INTRODUCTION

Since Kowalski's 1974 paper "Predicate Logic as Programming Language" [Kowalski 1974] there has been a growing interest in the use of what he calls "logic programming" as a technique for specifying computations. This interest has been well served by the PROLOG programming language, first implemented in 1972 at the University of Marseille. PROLOG supports a practical version of logic programming and has been in constant and growing use for over ten years.

The logic programming technique consists of formulating computational specifications as a set of declarative sentences, each of which is a simple assertion of some truth - conditional or unconditional, general or particular - which one wishes to record in a "knowledge base". The sentences are written formally as "Horn clauses".

A Horn clause has the form

    B if A1 and ... and An

in which B is the <u>conclusion</u> and the A's are the <u>conditions</u>. If n > 0 the Horn clause is "conditional", otherwise "unconditional".

Kowalski writes a conditional Horn clause as

    B <- A1 ... An

and an unconditional one as

    B <-

(We shall adopt a slightly more stylized notation for use in the computer.) Each of the A's and the B is an "atomic" sentence, i.e., a "predication", written as

    (P S1 ... Sk)

in which some predicate P is ascribed to a subject (S1 ... Sk). A subject is a tuple of descriptive expressions each of which is either a proper name, or a variable, or a "term", i.e., an applicative construction written as

```
        (F S1 ... Sn)
```

in which some operator F is applied to some operand (S1 ... Sn). The operand of a construction is in general a tuple of descriptive expressions of just the same kind as the subject of a predication.

A Horn clause containing one or more variables is general, while one containing no variable is particular.

We find it useful to make a slightly different classification of Horn clauses. An unconditional particular Horn clause is said to be a datum. Any other Horn clause is said to be a rule.

The variables in a general Horn clause are treated as if they were governed by universal quantifiers preceding the Horn clause. Thus, the Horn clause

```
   (Odd (Product x y)) <- (Odd x) (Even (Sum x y))
```

should be understood as being preceded by "for all x and y".

1.1 QUERIES AND ANSWERS

A knowledge base thus consists of rules and data.

For example, consider the knowledge base

```
      (Male James) <-
      (Male Bill) <-
      (Male George) <-
      (Parent Bill Mary) <-
      (Parent Mary James) <-
      (Parent George James) <-
      (Parent James Bill) <-
      (Father a b) <- (Parent a b) (Male a)
```

The first seven clauses are data; the eighth is a rule.

Once such a knowledge base is given, the logic programmer can request answers to queries. It is these requests which invoke the "logic computations" or deductions which reveal the implicit content of the knowledge base.

A query is a description

    the set of all (x1 ... xk) such that (A1 and ... and An)

of a set of tuples which satisfy a given conjunction (the constraint of the query). (Again, the notation used by our system is more "mechanical" than the informal style followed in this introduction.)

For example:

    the set of all (x y) such that (Father x z) and (Father z y)

The constraint of a query may contain variables in addition to those occurring in the answer template (x1 ... xk) of the query. These are to be understood as being governed by existential quantifiers preceding the constraint.

Thus the above query means

    the set of all (x y) such that
    there is a z such that
    (Father x z) and (Father z y)

The answer to such a query is then the set of all tuples whose satisfaction of the given constraint follows logically from the knowledge base. Thus the answer may be the empty set, or a set containing just one tuple, or a set containing many - even infinitely many - tuples. If the answer set is infinite, then in practice some finite subset of it will be supplied, or some other description of the set will be given.

Thus the answer to the above query would be

    { (James Mary), (George Bill) }

On the other hand the query

    the set of all x such that (Female x)

would have the answer

    { }

since our little knowledge base has no data or rules about the predicate "Female". Note that lack of information about "Female" does not cause an error message to appear. In practice, our logic programming system is prepared to report such "undefined predicates" as errors, and normally does so, but the programmer can easily suppress such reports.

A logic computation, then, consists of the sequence of events necessary to construct the answer to some query from the information embodied in some knowledge base.

1.2 PROLOG

These ideas were incorporated into the programming language PROLOG.

PROLOG implementations of logic programming go beyond the "pure" version of it described by Kowalski. They provide certain "imperative" features by which the programmer can affect the deductive computation of the answer to a query, and indeed by which he can affect the meaning of the query and of the assertions in the knowledge base.

These "control constructs" of PROLOG have been found most useful in practical applications of logic programming and we are in no sense critical of them. However, we believe that it is one of the essential ideas of logic programming to make a clean distinction between the "logic" of one's program and its "control".

1.3 LOGIC

Accordingly we have implemented a programming language called LOGIC, which embodies our idea of the "pure" version of logic programming featured in Kowalski's writings.

For those who may wish to avail themselves - while still in some sense working within a logic programming framework - of a greater degree of algorithmic control over events, we have embedded LOGIC within a system called LOGLISP.

1.4 LOGLISP = LOGIC + LISP

LOGLISP is a marriage of LOGIC with LISP.

A LOGLISP workspace contains everything one expects to find in a LISP workspace, and can be used purely as such by those who wish to ignore the presence of LOGIC in that workspace.

The same LOGLISP workspace can also be used as a "pure" LOGIC workspace, that is, as nothing but a basic logic programming environment, in which the assertion/query style of computing can be conducted in just the Kowalski manner. The logic programming facilities are invoked by making suitably-formed LISP calls on such LISP macros as ASSERT and the query macros ALL, ANY, THE, and SETOF. These LISP macros, together with further auxiliary and supplementary LISP macros and functions, comprise the LOGIC system.

A major advantage of embodying logic programming within LISP in this way is that the LISP environment is available to the logic programmer as a convenient host facility in which LISP functions for editing, displaying, monitoring, debugging, inputting and outputting one's assertions, queries and deductions can be invoked interactively or under program control.

Since the putting of a query is just the submission of an appropriate LISP function call, this can be done either (as in the PROLOG systems) interactively from the terminal or internally from within an applications program.

Since the answer to a query is a LISP data object it can either (as in PROLOG) be displayed on the terminal as a stream or returned to an internal call as its result and subjected, if desired, to analysis and manipulation.

Both predicates and operators in logic expressions can be given a LISP meaning by suitable programmer-supplied definitions of them as LISP function names. Some proper names indeed have a LISP meaning which is present in every workspace as part of LISP itself.

By a benign extension of the "pure" logic programming paradigm, LOGLISP is capable of recognizing such predicates and operators during the deduction cycle of LOGIC. The predications and constructions in whose heads they occur are thereby treated as LISP-meaningful function calls, and are replaced *in situ* by appropriate equivalents obtained by "reduction".

1.5 REDUCTION SEMANTICS VS. DENOTATION SEMANTICS

LISP users are accustomed to working with a "read-eval-print" loop at the top level of interaction with the machine. That is, the user types in an expression E and the machine prints out the object D which is denoted by E. The object D is constructed by *evaluating* E.

Thus, if the expression

```
        (+ (* 3 4) (* 4 5))
```

is typed in, the object

```
32
```

is printed. If the expression

```
(QUOTE (+ (* 3 4)(* 4 5)) )
```

is typed in, then the object

```
(+ (* 3 4)(* 4 5))
```

is printed.

In the first case, the E we enter and the D we get back are equivalent expressions. They both denote the same number. Some people (and indeed all LISP manuals) say that "32" *is* a number. In fact it is a *numeral* - an expression - not a number. You can print it, and numbers can't be printed. So in spite of the "official" story, in this first case E does not denote D, but rather both E and D denote the number thirty-two. E and D, that is, are *equivalent* expressions, and what LISP does is to take E and *reduce* it to D.

In the second case, however, LISP really does accept an E and produce the denotation D of E. Quotations really do denote what they quote. So in this case, E and D are *not* equivalent, and LISP is *not* simply reducing E to D.

In LOGLISP we have found it necessary to make LISP-reduction, rather than LISP-evaluation, the process that is applied to expressions when LISP is called from LOGIC. Ideally, we would implement a "read-reduce-print" loop at the top level, rather than the traditional "read-eval-print" loop, in order to have a more systematic LISP. However, we have taken it as one of our design principles that LOGLISP should merely extend, not modify, LISP as it stands.

One of the pleasant things about reduction is that it is always defined. For example, the expression

```
(+ (* 3  4) x)
```

reduces to

```
(+ 12 x)
```

instead of provoking an error message. Reduction consists of the persistent replacement of subexpressions according to "rewrite rules" which are either system-defined (as e.g. the rule "(* 3 4) = 12" defined by the multiplication operation) or user-defined (e.g. by naming and defining a function with LISP).

Reduction and instantiation ordinarily interact in a quite straightforward way. If we substitute 7 for x in "(+ (* 3 4) x)" we obtain the expression

```
(+ (* 3 4) 7)
```

which reduces to "19". If we make the same substitution in "(+ 12 x)" we obtain "(+ 12 7)", which also reduces to "19".

Reduction agrees with evaluation in the cases where the terminal expression is "an atom which denotes itself" - such as a numeral, or T, or NIL. It disagrees with evaluation in the cases (such as quotation) where evaluation of E produces a D which is not equivalent to E. LOGIC looks to LISP-reduction to transform predications and constructions into equivalent predications and constructions.

1.6 LOGIC CAN CALL LOGIC

The effect of this LISP-reduction step, performed once in every iteration of LOGIC's deduction cycle, is to give the LOGIC programmer the means to invoke very nearly the full power of LISP from within logic expressions.

This fact, together with the previously mentioned fact that LOGIC calls are simply certain LISP calls, means that it is very easy to initiate subordinate deductions during a deduction, by making recursive calls on LOGIC from within LOGIC.

Thus LISP is not only a rich and convenient host environment for LOGIC programming, but also a partner in the novel hybrid style of "LOGLISP" programming in which LISP and LOGIC call each other, and themselves, recursively.

The following chapters describe LOGLISP in full. The background ideas are explained in detail, and the design and implementation are presented both "top-down" and "bottom-up". Examples of applications of LOGLISP are given which illustrate its novel capabilities.

LOGLISP runs on the DEC-10 under the TOPS-10 operating system using a version of Rutgers-UCI LISP, essentially that described in [Meehan 1979], and on the LMI Lambda using Zetalisp [Moon and Weinreb 80].

CHAPTER 2

NOTIONS AND NOTATIONS.

In this manual we are concerned with computations whose data are expressions. It will be useful to have the basic ideas and notational conventions available from the outset, and in this chapter we discuss the most important of these. The general framework is that of LISP, augmented in certain ways to accommodate the needs of LOGIC.

2.1 EXPRESSIONS

LISP has two kinds of expression: atoms and dotted pairs. For the purposes of LOGIC we further divide the atoms into two kinds: variables and proper names. Therefore we have three kinds of expressions:

variables
proper names
dotted pairs

A variable is an identifier which begins with a lower case letter. (This is our usual convention. We allow the programmer to choose others when appropriate.) A proper name is any atom which is not a variable (in particular, strings and numerals are proper names).

A dotted pair is a composite expression with two immediate constituents, called its head and its tail, both of which are expressions. We have three formal predicates, for use in writing algorithms, which correspond to the three kinds of expression.

```
(VARIABLE u) = T if u is a variable, = NIL otherwise
(NAME u) = T  if u is a proper name, = NIL otherwise
(CONSP u) = T if u is a dotted pair, = NIL otherwise
```

We follow LISP's convention that truth is denoted by T, falsehood by NIL.

2.2 NOTATION FOR DOTTED PAIRS AND LISTS

We use the notation of LISP, which we review briefly here.

The dotted pair whose head is the atom A and whose tail is the atom 17 is written

```
(A . 17)
```

More generally, we express any dotted pair by writing its head, then a dot, then its tail, all enclosed in parentheses. Thus

```
((A . 17) . (B . C))
```

is the dotted pair whose head is (A . 17) and whose tail is (B . C).

A considerable notational economy is achieved by identifying certain expressions as lists and writing them without dots. All lists are dotted pairs except for one, which is the proper name: NIL. NIL is known as the empty list, and may also be written: (). Lists other than () are said to be nonempty. A nonempty list is any dotted pair whose tail is a list. A nonempty list may be written by writing its one or more components in order, with a left parenthesis before the first component and a right parenthesis after the last. The head of a list is its first component, and in general the (i + 1)st component of a list is the ith component of its tail. Thus the list

```
(0 . (2 . (4 . (6 . (8 . (BINGO . NIL))))))
```

has six components and would be written

```
(0 2 4 6 8 BINGO)
```

Note that the tail of a nonempty list is just the list of its remaining components after the head has been removed.

Both list- and dot-notations are blended together in the convention whereby, e.g.,

```
(A . (B . (C . (D . (E . (F . G))))))
```

can be written

```
(A B C D E F . G)
```

showing that it is like a nonempty list in having successive components, but unlike a list in that its "final tail" is not

NIL. In general, an arbitrary right-associated nest of dotted pairs

```
(x1 . (x2 . ... (xn . xn+1) ...))
```

is writable as the "dotted list"

```
(x1 x2 ... xn . xn+1)
```

and as the list

```
(x1 x2 ... xn)
```

in the special case that xn+1 is NIL.

2.3 NOTATIONS FOR LISP COMPUTATIONS

Although LISP expressions are of interest in themselves, LISP is a programming language in which certain expressions are interpreted as programs whose execution yields expressions as values. The most important of these are the applications. An application is a list whose head is an identifier, the name of a function, and whose remaining entries (if any) are expressions specifying the arguments of the function; the computation thus denoted is the application of the function named by the head to the values of the argument expressions.

To illustrate, the function which constructs a dotted pair from its two argument values is called CONS, and, e.g., the value of "(CONS 1 2)" is the pair (1 . 2). The decomposition functions are CAR, which yields the head of the pair which is its argument, and CDR, which yields the tail of its argument (both are undefined for atomic arguments). We have the fundamental identities

```
(CAR (CONS u v)) = u
(CDR (CONS u v)) = v
```

where u and v stand for any expressions.

When LISP expressions are interpreted as programs, numerals and strings are taken to be constants (i.e., to denote themselves) as also are the identifiers T and NIL. All other identifiers are variables, except when they appear as function names. We use quotations to denote other expressions. A quotation is a list of two components whose head is the identifier QUOTE and whose second component is an expression. The value of a quotation is its second component; thus the value of (QUOTE (A . B)) is the pair (A . B). Though standard, this notation is rather cumbersome and we follow the usual convention that, e.g., (QUOTE (A . 17)) may be abbreviated as '(A . 17).

This said, we can write

```
(CONS 1 '(2 3 4)) = '(1 2 3 4)

(CAR '(1 2 3 4)) = 1

(CDR '(1 2 3 4)) = '(2 3 4)
```

It is not mere pedantry that we have written quotations on the right of two of these equations. Both sides of such an equation are to be interpreted as "programs", and in writing such an equation we claim that the two sides have the same value. It would be incorrect to write

```
** (CONS 'A '(B C)) = (A B C) **
```

though the casual reader might overlook this slip, since A is not ordinarily the name of a LISP function. Contrast, though, the equation

```
(CONS 'CONS '(1 2)) = '(CONS 1 2)
```

which is correct, with

```
** (CONS 'CONS '(1 2)) = (CONS 1 2) **
```

which is quite plainly wrong. The value of the left side is the expression "(CONS 1 2)", while the value of the right is the expression "(1 . 2)".

Strictly as a matter of taste we shall write variables in LISP programs with lower case letters. This correct, though unorthodox, style is consistent with the conventions we shall introduce later for LOGIC programs.

Some additional functions will be used for computing with lists. The function which concatenates two lists is APPEND, defined by

```
(APPEND la lb) = if la is () then lb
                 else (CONS (CAR la) (APPEND (CDR la) lb))
```

Thus

```
(APPEND '(1 2 3) '(4 5 6)) = '(1 2 3 4 5 6)
```

The length of a list is the number of components it has:

```
(LENGTH lst) = if lst is () then 0 else 1 + (LENGTH (CDR lst))
```

2.4 PATHS. STRUCTURES. PRINTABLE EXPRESSIONS

The notion of a path is helpful in understanding the structure of expressions.

A path is a function obtained by composing CAR and CDR. The length of a path is the number of CAR's and CDR's in the composition. Thus CAR and CDR are the two paths of length 1. The four paths of length 2 are the functions

```
(CAAR u) = (CAR (CAR u))       (CDAR u) = (CDR (CAR u))
(CADR u) = (CAR (CDR u))       (CDDR u) = (CDR (CDR u))
```

Among the paths of length 3 is, e.g., the function

```
(CADDR u) = (CAR (CDR (CDR u)))
```

We spell the names of these function according to the usual LISP convention, beginning with C, followed by an A or a D for each CAR or CDR in the composition, and ending with R.

Thus CADDDAAADADAADADDDADADADADDDDDDAR is a path of length 31. This notation is quite general, but no actual LISP implementation known to us supports this spelling of paths for lengths greater than 4.

The identity function I is the (only) path of length 0.

An expression is said to admit a path p if the result of applying p to it is defined. Thus, every expression admits I, and every dotted pair also admits CAR and CDR. Variables and proper names admit only I, and this fact is their characteristic structural property. In general the set of all paths admitted by an expression exp is called the structure of exp, and gives a rather direct portrayal of exp's "shape".

A useful way to represent an expression is as a connected directed graph with two kinds of nodes - atoms and dotted pairs. A node which represents an atom has no out-arcs. A node which represents a dotted pair has exactly two out-arcs, one labelled CAR and the other labelled CDR. Each arc impinges upon exactly one node. Each node which represents an atom is labelled by the "printname" of that atom. There is a distinguished node called the root of the expression, from which there is at least one chain of arcs to every node in the expression. Each such chain beginning at the root node describes in the obvious way a composition of the functions CAR and CDR (the one obtained by reading the labels on the successive arcs of the path in reverse order). Such a graph G represents the expression exp if the paths admitted by exp are exactly those described by the chains of G, and if when (p exp) is an atom x, the chain describing p in G has x as its terminal node.

An expression may have many - even infinitely many - such representations as a graph.

Thus the expression whose head is 0 and whose tail is itself can be represented by the graph:

```
 ----------------
 |              |
 |         CDR  |
 -------->*------
          |
          |CAR
         \|/
          * 0
```

with two nodes, one of which is a dotted pair and the root, the other of which is the atom 0.

In such an expression-graph two chains are equivalent if they lead from the root to the same node. Thus in the above graph there are two equivalence classes of chains, namely those describing the paths in

{ I, CDR, CDDR, CDDDR, ... }

and those describing the paths in

{ CAR, CADR, CADDR, CADDDR, ... } .

This illustrates how in general the paths admitted by a given expression A are partitioned by each graph G which represents A into equivalence classes which correspond abstractly to the nodes of G. The class containing I always corresponds to the root. In general the system of equivalence classes shows how the structure of the expression is "shared" when represented by a graph. The same expression can have different sharing systems, corresponding to the different graphs which represent it. For example, the expression whose head is 0 and whose tail is itself can be represented by many other graphs, such as the (infinite) tree whose sharing classes are

{ I }, { CAR }, { CDR }, { CADR }, { CDDR }, ...

that is, all singletons. In this representation there is no sharing at all.

The printable expressions are those whose structure is finite. Not all expressions are printable. For example, the dotted pair whose head is 0 and whose tail is itself is not printable, since its structure is the infinite set of paths

{I, CAR, CDR, CADR, CDDR, CADDR, CDDDR, ... }

It may be described as the expression which solves the equation

x = (CONS 0 x)

and we may reason about it from this description. We may also represent it as a finite (cyclic) graph as discussed above. However, to attempt to print it would result in a nonterminating process.

2.5 ENVIRONMENTS

A dotted pair whose head is a variable represents a binding.

A list of such dotted pairs with distinct heads represents an environment.

Intuitively an environment is a collection of replacement instructions coded as dotted pairs, each one saying that a certain variable (its head) is to be replaced by a certain expression (its tail). An environment which contains all the bindings of the environment env (and perhaps other bindings) is called an extension of env.

2.6 THE NOTION DEF

If env is an environment and v is an expression we say that v is defined in env if, and only if, there is a binding in env whose head is v. Accordingly we introduce the function DEF by the scheme

```
    (DEF v env) = if v is () then NIL
                  else if (CAAR env) is v then T
                  else (DEF v (CDR env))
```

which computes the truth value that v is defined in env. Note that if v is defined in env then v is a variable.

2.7 THE NOTIONS IMM AND ULT

If v is defined in env we say that the immediate associate of v in env is the tail of the binding in env whose head is v, and we define the corresponding function IMM by

```
    (IMM v env) = if (CAAR env) is v then (CDAR env)
                  else (IMM v (CDR env))
```

with the understanding that IMM will never be invoked for a v and env such that v is not defined in env. The immediate associate in env of a variable v may itself be a variable defined in env. In such a case we may wish to track down the ultimate associate of v in env - namely the first expression in the series

```
          v , (IMM v env), (IMM (IMM v env) env) ...,
```

which is not defined in env. Accordingly we define the function ULT by

```
  (ULT v env) = if (DEF v env) then (ULT (IMM v env) env) else v
```

which computes, for any expression v and environment env, the ultimate associate of v in env. For example, if env is the environment

```
    ((x . y) (y . z) (z . (F v (B r s))) (r . (G s)) (s . 5))
```

then the immediate associate of x in env is y, but the ultimate associate of x in env is (F v (B r s)).

Note. Although we have defined DEF, IMM and ULT for arguments which are respectively a variable v and an environment env, it should be noted that all three functions work when v is any expression and env is any list of dotted pairs. This comment will be recalled later when we have defined the function UNIFY. End of note.

2.8 REALIZING EXPRESSIONS IN ENVIRONMENTS

Given an expression exp and an environment env, we consider the result of replacing each variable in exp by its immediate associate in env. This expression is called the realization of exp in env. To compute the realization of exp in env we use the function REAL, defined by:

```
(REAL exp env) = if (CONSP exp) then
                        (CONS (REAL (CAR exp) env)
                              (REAL (CDR exp) env))
```

```
                else if (DEF exp env) then (IMM exp env)
                else exp
```

We note that, for example, the realization of (+ x y) in the environment

```
    ((x . y) (y . z) (z . (F A (B r s))) (r . (G s)) (s . 5))
```

is (+ y z).

We are also interested in recursive realizations. For example, if we start with (+ x y) we obtain each of the following expressions by repeatedly realizing the previous one in the environment above:

```
        (+ y z)
        (+ z (F A (B r s)))
        (+ (F A (B r s)) (F A (B (G s) 5)))
        (+ (F A (B (G s) 5)) (F A (B (G 5) 5)))
        (+ (F A (B (G 5) 5)) (F A (B (G 5) 5)))
```

Realizing the final expression in this environment merely reproduces it. This final expression is therefore by definition the recursive realization of (+ x y) in the given environment. In general the recursive realization of an expression exp in an environment env is defined by:

```
(RECREAL exp env) = if (CONSP exp) then
                       (CONS (RECREAL (CAR exp) env)
                             (RECREAL (CDR exp) env))
           else if (DEF exp env) then (RECREAL (ULT exp env) env)
           else exp
```

2.9 UNPRINTABLE RECURSIVE REALIZATIONS OF PRINTABLE EXPRESSIONS

It can happen that a printable expression may have an unprintable recursive realization in a printable environment. For example, in the environment

```
                              ((x . (0 . x)))
```

the expression x has the recursive realization

```
                    (0 . (0 . (0 .      ...      )))
```

which is the "infinite expression" whose head is 0 and whose tail is itself.

2.10 UNIFICATION

A fundamental notion in logic programming is the operation of unifying two expressions expa and expb relative to a given environment env. This operation yields a result, denoted by (UNIFY expa expb env), which is either the message "IMPOSSIBLE" , indicating that expa and expb cannot be unified with respect to env, or else is an extension of env in which the recursive realizations of expa and expb are identical. In the latter case we say that the environment (UNIFY expa expb env) is the most general unifier ("mgu") of expa and expb with respect to env. By definition, we then have that

```
   (RECREAL expa (UNIFY expa expb env))
               = (RECREAL expb (UNIFY expa expb env))
```

The computation of (UNIFY expa expb env) is defined by

```
     (UNIFY expa expb env) =

      if env is "IMPOSSIBLE"  then  "IMPOSSIBLE"
      else (EQUATE (ULT expa env) (ULT expb env) env)
```

where

```
     (EQUATE expa expb env) =

      if expa is expb then env
      else if (VARIABLE expa) then  (CONS (CONS expa expb) env)
      else if (VARIABLE expb) then  (CONS (CONS expb expa) env)
      else if not (CONSP expa) then "IMPOSSIBLE"
      else if not (CONSP expb) then "IMPOSSIBLE"
      else  (UNIFY (CDR expa) (CDR expb)
                      (UNIFY (CAR expa) (CAR expb) env))
```

Note. If in the last line of the definition of UNIFY we replace the argument "env" by the argument

```
          (CONS (CONS expa expb) env)
```

we strengthen the unification algorithm considerably. It will be recalled that DEF, IMM and ULT are capable of accepting more general arguments, and of operating in effect as an associative retrieval system. When UNIFY is altered in this way, the "environments" are made to do double duty. We not only record bindings of variables in them, but also the pairs of expressions encountered in the final arm of the conditional - i.e., pairs which must be unified as part of the overall task. We are saying, in effect, that one of these expressions is to be replaced by the other if it should be encountered later. This change in the definition of UNIFY guarantees its convergence even on pathological cases involving complex infinite expressions. However, the extra overhead may be considered too great to warrant provision for such pathological cases. In LOGLISP we have implemented the algorithm essentially as given. <u>End of note</u>.

The mgu of (P (G x y) x y) and (P a (H b) c) with respect to the empty environment () is

```
((y . c) (x . (H b)) (a . (G x y)))
```

and in this environment both expressions are recursively realized as

```
(P (G (H b) c) (H b) c)
```

The mgu of expa and expb with respect to env is intuitively the most general way that env can be extended to an environment in which expa and expb can be recursively realized as identical expressions. It is possible that unifying expa and expb will make them unprintable. For example, the most general unifier of the expressions x and (0 . x) with respect to the empty environment () is the environment ((x . (0 . x))) in which x is bound to (0 . x). This shows that in general it is possible for (UNIFY expa expb env) to be an environment in which the recursive realizations of expa and expb are identical but unprintable.

2.11 SUBSTITUTIONS

Some readers may be more familiar with the usual treatment of unification, which is developed in terms of the idea of substitutions. A substitution is a mapping from expressions to expressions which preserves proper names and the dotted pair structure. More precisely, a mapping s from expressions to expressions is a substitution if, and only if, it satisfies the two conditions:

```
           p*s = p                 for all proper names p,
  (CONS x y)*s = (CONS x*s y*s)    for all expressions x and y.
```

We denote the result of applying a substitution s to an expression x by the notation: x*s, as illustrated above. An important property of a substitution is that its effect upon any expression is completely determined by its effect on the variables (if any) which it actually changes. By listing those variables, each equated to its image under the substitution, we therefore give a complete description of the substitution. But the information in such a list of equations is just what is provided by an environment. The list of equations

v1 = a1, ..., vn = an

corresponds to the environment

((v1 . a1) ... (vn . an))

and conversely. Indeed if s corresponds in this way to the environment env, then the image x*s of any expression x under s is just the expression (REAL x env). We write [env] for the substitution corresponding in this way to the environment env. Thus we have

x*[env] = (REAL x env)

for all expressions x and environments env. In this correspondence between environments and substitutions, the empty environment corresponds to the identity substitution (which transforms every expression into itself).

Composition of two substitutions sa and sb yields a substitution we denote by sa*sb (read "sa followed by sb") which sends each expression x into x*(sa*sb) = (x*sa)*sb. If sa is [enva] and sb is [envb], sa*sb is [envab], where envab is the list of all distinct bindings calculated by

(CONS v v*sa*sb)

where v is defined in enva or in envb (or both).

An environment env may be taken as a description not only of [env] but also of the iterate of [env]. The iterate s~ of a substitution s is the "limit" of the series

s, s*s, s*s*s, ...

To find the image x*s~ of an expression x under the iterate of s, we repeatedly apply s to x until no further changes occur. That is, x*s~ is the first expression in the series

x, x*s, x*s*s, x*s*s*s, ...

which is the same as its predecessor. It turns out that if s is [env] then x*s~ is (RECREAL x env). If s is [env] then s~ is denoted by {env}. So we have

x*[env] = (REAL x env)
x*{env} = (RECREAL x env)

Now in terms of substitution mappings, a unifier of two expressions expa and expb is a substitution s which maps expa and expb onto the same expression:

expa*s = expb*s

and a most general unifier of expa and expb is a unifier u of expa and expb with the property that

s = u*s

for all unifiers s of expa and expb.

Thus if u is an mgu of expa and expb and s is any unifier of expa and expb we have

expa*s = expa*u*s = expb*u*s = expb*s

and

expa*u = expb*u

so that the common expression onto which s maps expa and expb is obtainable by applying s to the common expression onto which u maps expa and expb. The substitution {(UNIFY expa expb env)} is the mgu of the two expressions expa*{env} and expb*{env}. Thus UNIFY is given the two expressions to be unified in an indirect way.

2.12 IMPLICIT EXPRESSIONS

The way that the two expressions expa*{env} and expb*{env} are given to the *UNIFY algorithm* is indirect, in "unassembled" form. This idea of working with expressions not yet (or possibly never) fully assembled is used extensively in our system. It makes for computational economy and also for increased intelligibility. We think of the list (expa env) as an "implicit" way of giving the expression expa*{env}. We say that expa is the skeleton part, and env the environment part, of the implicit expression (expa env). For many purposes it is more convenient, as well as more economical, to deal with such "implicit expressions" than with the actual expressions themselves. This is particularly the case when (expa env) describes an unprintable expression even though both expa and env are printable - as in the example previously mentioned when expa is x and env is ((x . (0 . x))).

2.13 INSTANCES

We often wish to consider, for some expression x, the various expressions x*s, where s is some substitution. These are known as the instances of x. For example, the expressions

```
(Divides 17 85)
(Divides (Plus a b) (Times 3 c))
```

are both instances of the expression (Divides p q) . The first of them is in fact a ground instance, since it contains no variables. In general we say that expressions which contain no variables are ground expressions: so a ground instance of x is an instance of x which happens to be a ground expression. Expressions which contain one or more variables are known as patterns. We often think of a pattern as a way of representing all of its instances.

2.14 VARIANTS

In the role of a representative of all its instances a pattern is not unique. Other patterns - known as its variants - have exactly the same instances. For example, the expressions

```
(Divides p q)    (Divides x y)
```

have exactly the same instances. Each is a variant of the other. In general a variant of an expression x is an instance x*s of x under a substitution which maps variables onto variables in one-to-one fashion. Such a substitution is called a variation, and is the only kind of substitution which has an inverse. If [env] is a variation then its inverse is [env'], where env' is obtained from env by interchanging the head and tail of each of its bindings. The compositions [env]*[env'] and [env']*[env] are then both the identity substitution.

In view of the identity of the set of instances of an expression with that of any variant of the expression, we often treat mutual variants as merely different ways of writing the same thing. However, in some of the computations involving patterns (such as the unification computation) it is sometimes necessary to take suitable variants of one's data beforehand.

To see why this is so, consider the problem of finding a pattern whose instances are exactly those which are instances of two given expressions, expa and expb.

For example, if expa and expb are the expressions

```
(Divides (Plus x y) z)        (Divides x (Times x y))
```

then among their common instances are the expressions

```
(Divides (Plus 3 4) (Times (Plus 3 4) 6))
(Divides (Plus 0 0) (Times (Plus 0 0)(Exp x y)))
```

and so on. We can get the first instance from expa by the substitution

```
x = 3, y = 4, z = (Times (Plus 3 4) 6)
```

We can get it from expb by the substitution

```
x = (Plus 3 4), y = 6
```

However, there is no single substitution s such that expa*s = expb*s = this common instance. The difficulty is the occurrence of the same variables in both expa and expb. If we take a variant of expb which has no variables in common with those of expa - say, the expression

```
(Divides p (Times p q))
```

which we shall call expc - then we can in fact find a pattern whose instances are exactly those common to expa and expb. To do this we need only compute the expression

```
(RECREAL expa (UNIFY expa expc ()))
```

or (which is the same)

```
(RECREAL expc (UNIFY expa expc ()))
```

which is the "most general common instance" of expa and expc - and therefore also of expa and expb.

Now the environment (UNIFY expa expc ()) is

```
((p . (Plus x y)) (z . (Times p q)))
```

and so the required expression is

```
(Divides (Plus x y) (Times (Plus x y) q) )
```

Every expression which is an instance both of expa and of expb is an instance of this expression - and conversely. This example illustrates the way in which the unification computation solves the general problem of constructing a pattern whose instances are precisely those which two given patterns have in common. Of course, when the two given patterns have no common instances, no such pattern exists. The UNIFY function detects all such cases by returning "IMPOSSIBLE" instead of an environment.

CHAPTER 3

LOGIC PROGRAMMING IN GENERAL

Logic programming is a "declarative" computing technique in which a program consists of one or more assertions. These assertions are used by the processor as "axioms" from which to deduce logical consequences.

Once such a set P of assertions has been installed, the processor is ready to evaluate expressions of the form

the set of all x1, ..., xn such that C

which in traditional mathematical notation is written

{ x1, ..., xn | C } .

Here, C is a sentence expressing a constraint which a tuple (x1, ..., xn) must be proved to satisfy, by a chain of deductive inference steps starting from P.

Such "set of" expressions are called <u>queries</u>. The result of evaluating a query { x1,...,xn | C } is a <u>set</u>

{ A1, ..., Ak }

of <u>answers</u> Ai, each answer being a tuple (t1, ..., tn) for which the processor can prove the sentence

C where x1 = t1 and ... and xn = tn

In logic programming no imperative constructs are used. The course of events during a logic computation triggered by a query Q is determined not by the programmer's control instructions (for there are none) but by the machine's pursuit of those deductive consequences of the program P which may yield answers to Q.

For example, the program might consist of the assertions stated in informal English in figure 1. These are numbered for later reference. Some of these sentences are "data" recording simple, particular facts; others are "rules" involving the use of logical variables x, y, z.

1 Drobny is a champion
2 Drobny is older than Rosewall
3 Rosewall is older than Goolagong
4 If x is older than y and y is older than z
then x is older than z
5 If x was born before y then x is older than y
6 Kelly is a child of Goolagong
7 If x is a child of y then y was born before x
8 Goolagong is female
9 Drobny is male
10 Rosewall is male
11 Rosewall is a champion
12 Goolagong is a champion
13 Connors is a champion
14 Borg is a champion
15 Connors is male
16 Borg is male
17 Borg was born before Connors
18 Connors was born before Kelly
19 Kelly is female
20 Evert is a champion
21 Evert is female
22 Evert was born before Connors

FIGURE 1

A logic programming system such as LOGIC is capable of constructing the set of all answers to a query about the "world" described by these assertions. In supplying the answers to such a query it must in general deduce them from what it has been told (rather than merely look the answers up). For example, the query:

```
the set of all x such that x is male
                           and x is a champion
                           and x is older than Kelly
```

would evaluate to the set of answers

{Connors, Borg, Rosewall, Drobny}.

That these persons are male and champions is explicitly given among the assertions, but that each of them is older than Kelly must be deduced. The deductions involved can, if desired, be examined by the user. For example, one could request an explanation of the fourth answer and LOGIC would respond with a rationale analogous to the informal explanation shown in figure 2 on the following page.

```
*   To show: Drobny is a male
             Drobny is a champion
             Drobny is older than Kelly

*   it is enough, by assertion 9,

*   to show: Drobny is a champion
             Drobny is older than Kelly.

*   But then it is enough, by assertion 1,

*   to show: Drobny is older than Kelly.

*   But then it is enough, by assertion 4,

*   to show: (there is a y:1 such that)
             Drobny is older than y:1
             y:1 is older than Kelly.

*   But then it is enough, by assertion 2,

*   to show: Rosewall is older than Kelly.

*   But then it is enough, by assertion 4,

*   to show: (there is a y:2 such that)
             Rosewall is older than y:2
             y:2 is older than Kelly.

*   But then it is enough, by assertion 3,

*   to show: Goolagong is older than Kelly.

*   But then it is enough, by assertion 5,

*   to show: Goolagong was born before  Kelly.

*   But then it is enough, by assertion 7,

*   to show: Kelly is a child of Goolagong.

*   But then it is enough, by assertion 6
    to show:  nothing.

*   End of explanation.
```

FIGURE 2

In the LOGIC system implemented within LOGLISP, the language of the queries, assertions and explanations is formalized and artificial. We shall shortly discuss the details of its design. Meanwhile, note that an explanation is essentially a proof, which proceeds in steps all of the same kind. At each step there is a "constraint list" of simple propositions, all to be shown true. Any variables in these propositions are considered to be existentially quantified by quantifiers placed at the beginning of the constraint list, and the constraint list itself is considered to be the conjunction of its members. The empty constraint list (i.e. the empty conjunction) is by convention true, so that if at some step the list has become empty, the proof is complete - there is nothing left to show. In general, each inference step consists of three stages:

(1) The selection of a proposition A from the constraint list and of an assertion from the knowledge base whose conclusion B will unify with A.

(2) The replacement of A in the constraint list by the constraints comprising the hypothesis (if any) of the selected assertion.

(3) The application to the new constraint list of the most general unifier of A and B.

The notion of unification has been defined only for formal expressions, however, and so to make this account precise we must now recast it in terms of the formal language of LOGIC. Let us now survey this formal language.

3.1 PREDICATIONS

The basic unit of the formal language is the <u>predication</u>. Predications are simple sentences of the subject-predicate form in which the predicate is written first and the subject second. The predicate P may be any "proper identifier" - that is, an identifier which is a proper name. (Recall that, in LISP, an identifier is an atom which is neither a numeral nor a string). The subject is a list of expressions called terms. Ground (i.e. particular) terms are essentially noun-phrases which denote things. A list (A1 ... An) of n ground terms denotes the n-tuple of things denoted respectively by the component terms A1, ..., An.

Predicates denote properties of tuples. (Properties of tuples are often also called relations.) The intuitive meaning of a ground predication with predicate P and subject A is the proposition that the tuple denoted by A has the property denoted by P. We write this formally as the list whose head is P and whose tail is A.

Thus we might formally write:

Drobny is a champion	as	(Champion Drobny)
Drobny is male	as	(Male Drobny)
Drobny is older than Kelly	as	(Older Drobny Kelly)
Evert is female	as	(Female Evert)
Evert was born before Kelly	as	(Before Evert Kelly)
Kelly is a child of Goolagong	as	(Child Kelly Goolagong)

3.2 TERMS

A term may be either a variable, or a proper name, or a construction. Constructions have an operator-operand form. The operator (which may be any proper identifier) denotes an operation, and the operand may be any list of terms. When the construction is a ground expression, its operand denotes a tuple of things, in just the same way as does the subject of a ground predication. Constructions are indeed syntactically indistinguishable from predications, and from LISP applications. Their common syntactic form reflects an underlying unity in their semantics as applicative expressions. We do not, however, require that predications and terms be meaningful LISP applications. Each ground construction or ground predication can be understood as representing the result of applying some function to some argument(s). In the case of a predication this means construing a property or relation as a truth function, namely a function which yields as its result one or other of the two truth values, TRUE, FALSE (T, NIL in LISP). We write the construction with operator F and operand (A1 ... An) as the list

(F A1 ... An)

whose head is F and whose tail is (A1 ... An).

Ground predications, then, express facts and denote truth values. Ground terms express applicative descriptions and denote things. Both ground terms and ground predications have the same simple, systematic denotational semantics based on the applicative principle.

3.3 WORLDS

A world is a collection of facts - "everything that is the case" in that world. In logic programming a world is represented by a collection of ground predications.

Given a collection W of ground predications as such a world, we can ask for what substitutions, if any, a given predication Q (whether ground or not) is "true in W".

If Q is a ground predication, this is simply the question whether Q is a member of W. If Q is in W, the answer is then: the identity substitution.

If Q is a predication <u>pattern</u>, however, this is not quite so simple a question, and we construe it to mean: for which substitution operations s is the predication Q*s in W?

For example, the world specified by the assertions in our earlier example is the set shown in figure 3.

With this world as W, if we ask what are the substitutions for which the predication

(Male x)

is true in W, we get four "solutions", namely:

x = Drobny
x = Rosewall
x = Borg
x = Connors

there being four ground instances of "(Male x)" in W, namely those corresponding to these four substitutions. More generally we can ask a question involving a conjunction of predications.

```
(Male Drobny)      (Female Goolagong)   (Champion Drobny)
(Male Rosewall)    (Female Evert)       (Champion Rosewall)
(Male Borg)        (Female Kelly)       (Champion Borg)
(Male Connors)                          (Champion Connors)
                                        (Champion Goolagong)
                                        (Champion Evert)
(Older Drobny Rosewall)
(Older Drobny Goolagong)                (Before Borg Connors)
(Older Drobny Kelly)                    (Before Connors Kelly)
(Older Rosewall Goolagong)              (Before Evert Connors)
(Older Rosewall Kelly)                  (Before Goolagong Kelly)
(Older Goolagong Kelly)
(Older Borg Connors)
(Older Borg Kelly)
(Older Evert Connors)                   (Child Kelly Goolagong)
(Older Evert Kelly)
(Older Connors Kelly)
```

FIGURE 3

If Q1, ..., Qn are predications, we can ask of a world W

```
for what substitutions s
is (Q1 & ... & Qn)*s true in W?
```

or more briefly:

```
what substitutions satisfy (Q1 & ... & Qn) in W?
```

For example in the W of our example the question

```
what substitutions satisfy
 ((Male x) & (Champion x) & (Older x Rosewall))
in W?
```

has the answer

```
x = Drobny
```

since under this (but no other) substitution the conjunction becomes true in W.

3.4 QUERIES

In LOGIC we write the query

```
the set of all X such that Q1 and ... and Qn
```

formally as an expression of the form

```
        (ALL X Q1 ... Qn)
```

in which Q1 ... Qn are predications and X is an expression called the answer template of the query. The answer template may be any variable, any proper name, or any list of terms. The list Q = (Q1 ... Qn) is the constraint list of the query.

For any world W, such a query has a set of answers, which is represented as a list of expressions. Each expression in this "answer list" is the instance of the answer template under a substitution which satisfies the constraint list Q, that is, which transforms the conjunction (Q1 & ... & Qn) into one which is true in W. Thus the query

```
         (ALL x (Male x)
                (Champion x)
                (Older x Rosewall))
```

has the answer list (in the world of our example)

```
             (Drobny)
```

since the substitution x = Drobny is the only one which satisfies the given constraint, while the query

```
      (ALL z (Female z) (Older z Drobny))
```

has the empty list

```
             ()
```

as its answer list since there are no substitutions which satisfy the constraint

```
       ((Female z) (Older z Drobny))
```

3.5 SPECIFYING A WORLD BY ASSERTIONS

It is not expected that one should have to specify a world by explicitly listing, as in figure 3, all of its predications (although this would in principle be possible for a finite world). A world is specified indirectly, by giving a collection of <u>clauses</u>. A clause is a sentence with two main parts: a <u>conclusion</u>, which is a predication, and a hypothesis, which is a list of predications. The hypothesis of a clause can be the empty list, in which case the clause is said to be an unconditional clause, whereas a clause whose hypothesis is nonempty is said to be a conditional clause. An unconditional clause whose conclusion is B is asserted by the command

```
        (ASSERT B)
```

while a conditional clause with conclusion B and hypothesis (A1 ... An) is asserted by the command

```
        (ASSERT B <- A1 & ... & An)
```

(the arrow and the ampersands are optional "syntactic sugar" and may be omitted).

A collection of clauses is called a knowledge base. Any such collection determines a world.

An unconditional ground clause (i.e. a datum) asserted by (ASSERT B) intuitively says that B is one of the facts in the world being described - "B is true". Recall that any clause which is not a datum is a rule. A rule asserted by the command (ASSERT B <- A1 & ... & An) says that B is one of the facts in the world being described provided that A1,...,An all are - "if A1 and ... and An are true then B is true". A rule which is a clause pattern - a clause containing one or more variables - has the same descriptive effect as would the set of all its ground instances. In general this means that a clause pattern is in effect a universally quantified statement. If its variables are x1,...,xk (say) then the clause asserted by (ASSERT B <- A1 & ... & An) can be read

"for all x1, ..., xk: if A1 and ... and An then B"

Indeed, if some of the variables among the xi (say, z1,...,zp) do not occur in the conclusion B while the rest (say, y1,...,yt) do, the clause asserted by (ASSERT B <- A1 & ... & An) may be more intuitively (but equivalently) read

```
   "for all y1, ..., yt:
      if  there exist z1, ..., zp such that A1 and ... and An
      then B"
```

In the example of figure 1 there are three such clause patterns. All the other clauses in figure 1 are data. Figure 4 shows the series of commands which would set up the knowledge base of figure 1, numbered to correspond with figure 1. The numbers would not be typed when entering these commands into the computer.

```
 1   (ASSERT (Champion Drobny))
 2   (ASSERT (Older Drobny Rosewall))
 3   (ASSERT (Older Rosewall Goolagong))
 4   (ASSERT (Older x z) <- (Older x y) & (Older y z))
 5   (ASSERT (Older x y) <- (Before x y))
 6   (ASSERT (Child Kelly Goolagong))
 7   (ASSERT (Before y x) <- (Child x y))
 8   (ASSERT (Female Goolagong))
 9   (ASSERT (Male Drobny))
10   (ASSERT (Male Rosewall))
11   (ASSERT (Champion Rosewall))
12   (ASSERT (Champion Goolagong))
13   (ASSERT (Champion Connors))
14   (ASSERT (Champion Borg))
15   (ASSERT (Male Connors))
16   (ASSERT (Male Borg))
17   (ASSERT (Before Borg Connors))
18   (ASSERT (Before Connors Kelly))
19   (ASSERT (Female Kelly))
20   (ASSERT (Champion Evert))
21   (ASSERT (Female Evert))
22   (ASSERT (Before Evert Connors))
```

FIGURE 4

The knowledge base set up by the commands of figure 4 completely determines the world of figure 3, according to the following general definition.

DEFINITION

The world determined by a knowledge base D is the smallest set W of ground predications which satisfies the two conditions:

(1) if D contains the datum G, then G is in W

(2) if G is a ground instance of a rule in D and the predications in the hypothesis of G are all in W, then the conclusion of G is in W.

END OF DEFINITION

In effect, this definition describes a process which infers W from D by a series of wholesale inference steps. First, by (1), the process constructs outright the set W0, which contains just those ground predications which are conclusions of data in D. Then by (2), in general, having constructed the set Wn, this process constructs Wn+1 by adding to Wn the conclusion of every ground instance G of every rule in D, provided that every predication in the hypothesis of G is in Wn. Thus the process constructs a series of bigger and bigger worlds

W0, W1, ..., Wn, ...

which either ends (with a world that is the same as its predecessor) or else continues indefinitely. The world W is then the "limit" of this series, i.e., the union of all of the sets in it , i.e. the smallest set which includes them all.

Thus the world W is determined by a knowledge base D through a "bottom up" process of reasoning.

Given such a D, we wish to be able to answer queries about its world W. In doing so we wish to avoid the brute force method of generating W bottom up and searching it. It is much better, given a query about W, to reason "top down" about W's contents without actually constructing W. This turns out to be possible through the use of unification, built into a special inference principle called LUSH resolution. This inference principle can be applied very efficiently through the use of implicit expressions, as we shall now see.

3.6 IMPLICIT CONSTRAINTS AND THEIR SOLUTIONS

By an implicit constraint we mean a list (q env) in which env is an environment and q is a list of predications. The expression q*{env} is the corresponding explicit constraint . Now let D be a knowledge base and let W be the world described by D. We denote by (SOL q env D) the set of "solutions of (q env) in D" - that is, the set of environments xenv which are extensions of env with the property that all of the predications in q*{xenv} are true in W.

We wish to calculate (SOL q env D) from (q env) and D.

There are two cases to consider. The first case is when q is empty. Then (SOL q env D) is simply the set whose only member is env. Such a (q env) is said to be solved.

The second case is when (q env) is unsolved, i.e., when q is nonempty.

For this case we use LUSH resolution to represent the desired set as the union of one or more simpler sets.

3.7 LUSH RESOLUTION

For any unsolved constraint (q env), any knowledge base D, the set

(RES q env D)

is a set (possibly empty) of implicit constraints called the D-resolvents of (q env). The interest of this set lies in the fact that we have:

(SOL q env D) = (SOL q1 env1 D) U ... U (SOL qn envn D)

where (q1 env1), ..., (qn envn) are the D-resolvents (if any) of (q env). In particular it may be that there are no D-resolvents of (q env). This then means that there are no solutions of (q env) in D.

3.8 SEPARATION OF VARIABLES

The computation of (RES q env D) requires the determination of a variant D' of the knowledge base D. D' must have the property that none of its clauses contains a variable which occurs in (q env). This "standardizing apart" of the variables in the constraint from those in the clauses is necessary for the theoretical completeness of the resolution transformation. In the current implementation D' is selected automatically and represented implicitly and economically by techniques explained in [Robinson-Sibert 1984].

3.9 DEFINITION OF (RES Q ENV D)

The set (RES q env D) is the set of all implicit constraints calculated as

(CONS (APPEND h (CDR q)) (UNIFY (CAR q) c env))

for which h is the hypothesis of a clause in D' whose conclusion c unifies with (CAR q) in env.

The decision to unify (CAR q) with c is entirely arbitrary; one could equally well choose some other predication of q. Although a well-informed choice might offer substantial benefits to the overall computation, we know of no economical way to make such a choice, so the present implementation uses the simplest method available.

3.9.1 The Computation Of (RES Q Env D)

On the face of it, the entire knowledge base D must be searched in order to extract from it every clause whose conclusion c will unify in env with the predication p = (CAR q).

Fortunately, this is not necessary. For large D the cost would be prohibitive.

In fact it is possible to store D in such a way that only a relatively small subset of D need be searched. Note, first, that the predicate of c must be the same as that of p if c is to unify with p in env. Accordingly, only those clauses need be considered whose conclusions satisfy this condition, and it is straightforward to partition D into subsets of clauses ("logical procedures") whose conclusions have the same predicate. Each logical procedure can be stored on the property list of the predicate of its conclusion, and thus be retrievable in time essentially independent of the size of D. The data of each procedure can be further indexed on the basis of the various proper identifiers which occur in their conclusions. This is highly advantageous, since in order that a datum c unify with p in env, c must in fact contain every proper identifier which occurs in p*{env}. This observation forms the basis of a quite selective retrieval technique. In practice it is found that large procedures consist mainly, if not entirely, of data, so that the retrieval technique frequently applies just when it will do the most good.

3.10 THE DEDUCTION CYCLE

The heart of the LOGIC system is the basic deduction cycle, which computes the set (SOL q env D) for a given implicit constraint (q env) and a given knowledge base D.

The computation of (SOL q env D) consists of the development of two sets of implicit constraints, SOLVED and WAITING. Initially, SOLVED is empty and WAITING contains the single constraint (q env). These two sets are then subjected to an iterative transformation which corresponds intuitively to the construction of a "deduction tree" whose nodes are implicit constraints. The root of this tree is the implicit constraint (q env). The successors (if any) of an unsolved node (x e) are the D-resolvents of (x e). The tips of the deduction tree are the solved nodes (if any) and the unsolved nodes (if any) which have no D-resolvents. The output of the deduction cycle is the set of environment parts of the solved nodes of the tree.

As the tree develops, the solved nodes are collected into the set SOLVED, and the nodes which have not yet been processed are kept in the set WAITING. Thus the tree construction is finished when WAITING finally becomes empty.

The deduction cycle is the following three-step algorithm:

```
  IN:  let SOLVED be the empty set and
       let WAITING be the set containing only (q env)

 RUN:  while  WAITING is nonempty

       do  1  remove some constraint (x e) from WAITING

           2  if    (x e) is solved
              then  add (x e) to SOLVED
              else  add the D-resolvents of (x e) to WAITING

 OUT:  return the set of environment parts of SOLVED
```

In general (SOL q env D) is computed by executing the deduction cycle and taking its output as the required set.

Several points are worth noting about the deduction cycle.

3.10.1 Failure Nodes: Immediate And Ultimate

An unsolved node which has no solved nodes as descendants is known as a "failure". There are two kinds of failure. An immediate failure has no descendants at all - because it has no D-resolvents. An ultimate failure has one or more successors, but they too are failures - the entire subtree rooted in an ultimate failure consists of nothing but failures, and its tips are all immediate failures. It is an interesting problem to design implementations of the deduction cycle in which the subtrees rooted in ultimate failures are kept as small as possible without undue extra computation. Ideally, all failures would be immediate and would be recognised as such in constant (and short) time.

3.10.2 Nondeterminacy Of Deduction Cycle

The choice called for in step 1 of the deduction cycle introduces an element of nondeterminacy. The choice can be made uniformly and cheaply according to a criterion which is built into the system design. In the PROLOG systems, the selection in step 1 is in effect ruled by a very simple criterion - the first constraint (x e) is selected from a WAITING which is represented in effect as a list. [We have to say "in effect" because in fact the PROLOG systems handle WAITING dynamically in a backtrack mode of working which never explicitly realises the whole list at once.]

The selection of the node (x e) in step 1 can (as in the PROLOG systems) be made according to the "depth first" criterion in which the younger members of WAITING are chosen before the older members. This may sometimes lead to the "depth first runaway" situation in which one or more nodes in WAITING are never selected because they are never the youngest. In practice other considerations (see the discussion below of the deduction window) preclude an infinite depth first runaway, but even the finite versions of it which are allowed by the deduction window may be thought undesirable. Avoidance of depth first runaway can be economically achieved by letting the selection in step 1 depend upon a quantity which can be computed once for all for each node when it is first generated. This quantity is the "solution cost" of the node.

3.10.3 Definition Of Solution Cost

The solution cost of a node (x e) is simply a heuristic estimate of the "cost" (in arbitrary units) of obtaining a solved descendent of (x e). In LOGIC we estimate this cost as the sum of (LENGTH x) and the depth of (x e), which is number of nodes preceding (x e) on its branch of the deduction tree. The simplest heuristically guided search results from selecting in step 1 a node of WAITING having minimum solution cost. Our actual search technique combines this method with a limited depth first search; the details are explained in chapter 9.

3.11 THE DEDUCTION WINDOW

Since in general the deduction tree can be infinite, it is possible that WAITING should always be nonempty, and hence it is necessary in these cases to truncate the deduction cycle and accept the resulting (perhaps incomplete) set of solutions as an approximation to the full set (which may be infinite).

It is desirable to manage this truncation gracefully and to provide the LOGIC user with some control over its details. This is the reason for the deduction window.

The deduction window is a collection of parameters which can be set in various ways by the user and which have default values which are used in the absence of user-provided alternatives.

The deduction window is discussed in more detail in Chapter 9.

Each parameter in the deduction window is used as an upper bound on an associated quantity measuring some feature of the deduction cycle. These quantities are TREESIZE, NODESIZE, ASSERTIONS, RULES and DATA.

At a given moment in the execution of the deduction cycle TREESIZE is the total number of nodes which have so far been generated. The RUN loop is terminated as soon as TREESIZE exceeds the bound set for it in the deduction window.

The implicit constraint (x e) selected in step 1 of the body of the RUN loop is treated as an immediate failure (hence dropped from WAITING without progeny) if NODESIZE(x e), ASSERTIONS(x e), RULES(x e) and DATA(x e) are not all within the bounds specified for them in the deduction window.

NODESIZE(x e) is (LENGTH x), the number of predications in the constraint list X of (x e).

ASSERTIONS(x e) is the number of nodes which precede (x e) on the branch of the deduction tree of which it is the current tip. This number is the same as the number of clauses invoked in its deduction. It is 0 for the initial node, and is 1 greater than that of its predecessor for each derived node.

RULES(x e) is a quantity similar to ASSERTIONS(x e), but reflects the classification of clauses into rules and data.

RULES(x e) is the number of times a rule was invoked in the deduction of (x e), and

DATA(x e) is the number of times a datum was invoked in its deduction. We obviously have, for each (x e) in WAITING, that:

DATA(x e) + RULES(x e) = ASSERTIONS(x e) .

Thus the deduction window serves as a truncation device which ensures that each particular execution of the deduction cycle will terminate. It provides the user with both a global (TREESIZE) and a local (NODESIZE, ASSERTIONS, RULES and DATA) cutoff control. All the bounds in the deduction window are set to system defined default values in the absence of user-defined alternatives.

CHAPTER 4

LOGIC PROGRAMMING IN LISP

LOGIC is related to LISP in two different ways.

First, it is implemented in LISP - that is, the LOGIC system consists of a collection of LISP functions which live in a LISP workspace and provide all the logic programming facilities described in this manual.

Second, LOGIC in a certain sense contains LISP. This means that the LOGIC programmer can invoke LISP from within LOGIC calls, by incorporating, in clauses and queries, pieces of text which can be handed over to LISP for processing. To understand how this works we need to discuss the notion of LISP-reduction.

4.1 LISP-REDUCTION OF LOGIC EXPRESSIONS

The expressions encountered by the LOGIC "processor" during the deduction cycle are terms and predications arising ultimately from the constraint list of some query and from the clauses used in constructing resolvents. However, some of these LOGIC expressions may also admit an *interpretation* as LISP programming constructs. In that case they may have a LISP value, or if not they may be capable of some LISP-reduction.

For example, the expression

```
(+ 3 (* 5 4))
```

is both a LOGIC term and a LISP construct. In the latter role, it is equivalent to, and can be replaced by, its "value", namely the numeral

```
23
```

within any ordinary expression e to produce an expression which has the same meaning as e. Both expressions denote the number twenty-three.

Such replacements of expressions by others which are their values are basic equivalence-preserving transformations of ordinary computation as normally conceived. The presence of free variables does not invalidate this idea. Thus even though "a" has no LISP value, the LISP construct

(+ a (* 5 4))

can be reduced; it is LISP-equivalent to and can be replaced by the simpler expression

(+ a 20)

even though the latter is not its "value" as in the first case. In general, an expression may well "reduce" to another expression even when it will not, in the usual sense, "evaluate" to a "value".

We refer to this process of replacing a LOGIC expression by one which is LISP-equivalent to it as "LISP-reduction", or simply as "reduction" when this will not cause confusion. It can be done to any expression at any time and is always defined (but may be merely the identity transformation). When an expression reduces only trivially, i.e., to itself, we say that it is "reduced".

4.2 LISP DEFINITIONS

Certain definitions are built into LISP itself and come with the system whenever one sets up a LISP workspace. That is, certain identifiers are defined as denoting built-in LISP functions (CAR, CDR, PLUS, etc.) or as the keywords of built-in special forms (COND, SETQ, PROGN, etc.).

In addition to these built-in LISP definitions, a LISP workspace may contain further definitions made by the user. A collection of such user-coined LISP definitions indeed constitutes a LISP program.

4.3 REDUCTIONS AND VALUES.

The joint effect of the system- and user-imposed definitions in a LISP workspace is to determine a notion of "reduction".

Every LISP construct is reducible, if only trivially (to itself). The reduction process produces (intuitively) a "reduction series"

C0, ..., Cn

of LISP expressions, in which C0 is C itself, and Ci+1 comes from Ci by the replacement of some subexpression R by an equivalent expression S. We think of this as the invocation of the "rewrite rule"

    R = S

as for example the rule

    (+ 3 4) = 7 .

We say that Ci is "rewritable", and "rewrites to" Ci+1. Thus a reduction series consists of one or more expressions the last of which (if the series terminates) is not rewritable, but each earlier expression (if there are two or more) rewrites to the next.

It is in the nature of the concept of reduction that a reduction series is continued as far as possible, i.e., until an expression Cn is reached which is not rewritable. Such unrewritable expressions are often said to be "in normal form" or "normal". As we said above, usage also sanctions calling them "reduced".

[There are expressions which cannot be reduced to normal form, because it is always possible to apply further rewrite rules. For example, if the only rule is

    x = (F x)

then the reduction series for x is

    x, (F x), (F (F x)), ...,

and so x does not have a normal form.]

Thus reduction is always defined. It often coincides with evaluation - that is, the value of e and the reduction of e are often identical. But this is not always the case and the matter requires some care.

For example, the quotation

    '(This is an S-expression)

has as its value the expression

    (This is an S-expression)

but it is reduced (as are all quotations), that is, it is its own reduction.

The expression

```
(* (+ 3 4) (// 5 x))
```

has no value (since its second argument expression contains an occurrence of a variable) but reduces to the expression

```
(* 7 (// 5 x))
```

These two examples show that although an expression always has a reduction it may or may not have a value, and that even when it does have a value, this may or may not be the same as its reduction.

The following two propositions hold in general:

A. If an expression has a value which is a proper name, its reduction is that proper name.

B. If an expression e has a value v which is not atomic, or is a variable, the reduction of e is the expression (QUOTE v), rather than the expression v.

Proposition B is at first a somewhat surprising feature of the reduction notion. A little reflection soon shows its naturalness.

By definition, the reduction of an expression is always a reduced expression. Moreover, an expression e must be LISP-equivalent to the reduction of e - and this means that if e has the value v so must the reduction of e. These two considerations together require that the reduction of e be (QUOTE v) - the value of which is v - since the expression v might itself have a value w distinct from v. Only when v is a proper name is w identical with v.

Note that one effect of these definitions is to establish a convention for quoting atoms which differs somewhat from that used in LISP. As an example, the LOGIC expression

```
(MEMBER Borg '(Connors Borg Evert))
```

has value (Borg Evert), being analogous to the LISP expression

```
(MEMBER 'Borg '(Connors Borg Evert)) .
```

The utility of the LOGIC convention becomes apparent when one considers a predication such as

```
(Older Drobny Rosewall)
```

which, had LOGIC followed the LISP convention, would have to be written

(Older (QUOTE Drobny) (QUOTE Rosewall)) ,

a rather less palatable form.

4.4 OBJECTS IN LOGLISP

Before proceeding into a detailed exposition of the interaction between LOGIC and LISP, we review the classification of LISP objects imposed by LOGIC. Recall that an object is either atomic or composite. Atoms are identifiers, strings or numerals. Identifiers beginning with a lower case letter are variables, all others are proper identifiers. Proper identifiers, strings and numerals constitute the class of proper names. For technical reasons, we prohibit the use of the character "^" in variables, except for certain "subscripted variables" created by LOGIC, which will be explained later.

4.5 REDUCTION AND EVALUATION

Generally speaking, the "applicative" expression e = (f e1 ... eN) has a value if f is the name of a function (defined in LISP) and e1, ..., en have values for which f is defined. In this case the reduction of e is the value of e, quoted when necessary as explained above. [The value of (f e1 ... eN) is obtained by APPLYing f to the values of the expressions ei].

The reduction of an applicative expression which does not have a value is in general obtained by replacing occurrences of its immediate subexpressions by occurrences of their reductions.

We proceed now to a precise definition of the notion of reduction.

We shall speak of expressions as though they were explicitly represented. In fact, in the LOGLISP system we compute the reduction of an expression directly from its implicit representation, as economically as we can. The resulting reduction is also represented implicitly, with the same environment part.

4.5.1 Expressions And Their Values

Excepting certain special forms which are discussed below, we say that the expression e = (f e1 ... eN) has value v if (AND (SYMBOLP f) (FBOUNDP f)), the contents of f's function cell (fc) is a lambda expression, e1, ..., eN have values, and v is the result of APPLYing fc to the values of e1, ..., eN.

NOTE: Currently, LOGLISP will not properly evaluate expressions of the form (f e1 ... eN) where f is a user defined special form. (A user defined special form is a function whose formal arglist contains one or more of the keywords &QUOTE, &REST, &KEYWORD, or &EVAL.)

The value of a proper name is the proper name itself.

A variable has no value.

4.5.2 Expressions And Their Reductions

Again with the exception of certain special forms, an expression e of the form (f e1 ... eN) has the reduction r if

(a) e has the value v, in which case r is v, if v is a proper name; otherwise r is (QUOTE v)

or e has no value, but

(b) f is a proper identifier, in which case r is (f e1' ... eN'), where ei' denotes the reduction of ei.

(c) otherwise, r is e itself.

Note that atoms, whether variables or not, are reduced.

Note further that every expression (f e1 ... eN) in which f is a variable, a number, or, indeed, anything except a proper identifier, has no value, and is reduced. This may be justified intuitively on the ground that one doesn't know what to do in such a case. We could, in fact, have extended the notion of reduction to allow f to be a lambda expression, say, but we have not chosen not to do so. Such an extension would have complicated matters significantly with no great advantage in flexibility.

4.5.3 Side-effects

Reducing an expression which has a value and whose (LISP) evaluation produces one or more side effects causes those effects. Note that the reduction of such an expression is either a proper name or a quotation -- an expression which has the same value but whose evaluation produces no side effects. To illustrate, reducing

```
(SETQ A (+ (SETQ B (* 2 3)) x)
```

yields the expression

```
(SETQ A (+ 6 x))
```

having assigned 6 as the (LISP) value of the identifier B. If this expression is later instantiated to

```
(SETQ A (+ 6 (* 4 5)))
```

and that expression reduced, the reduction is 26, with the effect of assigning 26 to the identifier A. Observe that such effects may be separated in time, owing to the non-deterministic nature of the search performed by LOGIC. The computation of the reduction of any one expression is, on the other hand, an "atomic" act within this search, no matter how complex the expression, and any effects will occur in the order one would expect in LISP evaluation.

Run-time error messages are a particular kind of side-effect which may arise while reducing an expression. When they arise, these are produced by the LISP interpreter, and may be dealt with in the usual ways provided by LISP, as well as by some additional means provided with LOGLISP and described in Chapter 12. As an example, an attempt to reduce

```
(+ A 2)
```

causes the computation to be broken with a "NON-NUMERIC ARGUMENT" message, since the proper name A has itself as value, and + requires that its argument values be numbers. One might suppose that such an expression should be regarded as having no value, and hence reducing to itself, but to adopt such a policy in the present implementation would be quite impractical.

4.6 SPECIAL FORMS

In addition to the expressions just considered there are several special forms which require separate treatment. Most of these are special forms of LISP.

Since the syntax of special forms is the same as that of applicative forms whose function designator is atomic, LISP users often slur over the distinction. It is, however, most important to remember that the LISP value of a special form is NOT obtained by "applying the function denoted by its head to the object denoted by its tail" - that being how the LISP value of an APPLICATIVE form is obtained.

There is a special process set up for obtaining the LISP value of each special form, to which a LISP interpreter switches on recognizing the keyword (COND, SETQ, PROGN, QUOTE, etc.) of that special form.

This little homily would not be necessary if the syntax of applicative forms were designed in the same way, and applicative forms were tagged as such by a keyword, say, APP. The high frequency of applicative forms in programs would make such a convention burdensome. No one wants to have to write

```
(APP + (APP * 3 4) (APP SIN 30))
```

instead of

```
(+ (* 3 4) (SIN 30))
```

4.6.1 Macros

Let (FSYMEVAL f) be a macro definition fm. The expression e = (f e1 ... eN) is first reduced to the macro expansion of e as defined by fm. The expression e has a value only if the macro expansion of e has a value.

Some examples: (in the context of the following macro definition)

```
(DEFMACRO M (X Y) `(+ (* ,X ,X) ,Y))
```

```
(M 2 x) is reduced to (+ 4 x).
(M (+ 1 2) 5) evaluates to 14.
(M (+ 1 x) (+ 1 2)) reduces to (+ (* (+ 1 x) (+ 1 x)) 3).
```

4.6.2 Quotations

(QUOTE v) or (FUNCTION v)

Each of these forms is reduced.

Each has the value v.

Each of these forms is "immune" to instantiation, that is, (QUOTE v)*s is (QUOTE v), for any substitution s, even though v*s may be different from v.

(F-L args . exprs)

This form is reduced, and, like the QUOTE and FUNCTION forms, it is immune to instantiation. See [Meehan 1979, p.57] for details on the use of F-L in LISP.

4.6.3 Listings

(LIST e1 ... eN)

(LIST) has the reduction NIL.

If e1, ..., eN have the values v1, ..., vN then (LIST e1 ... eN) has the reduction (QUOTE (v1 ... vn)).

If not all of the ei's have values, then (LIST e1 ... eN) has the reduction (LIST e1' ... eN'), where ei' denotes the reduction of ei.

This is just what one one would expect.

4.6.4 Conjunctions

(AND e1 ... eN)

(AND) reduces to T.

(AND e) reduces to the reduction of e.

If e1 has the value NIL then (AND e1 ... eN) reduces to NIL.

If e1 has a non-NIL value then (AND e1 ... eN) reduces to the reduction of (AND e2 ... eN).

If e1 has no value then (AND e1 ... eN) reduces to (AND e1' e2 ... en), e1' being the reduction of e1.

All of this corresponds to LISP usage, the conjuncts being taken in order and only as far as necessary to determine the result.

4.6.5 Disjunctions

(OR e1 ... eN)

(OR) reduces to NIL.

(OR e) reduces to the reduction of e.

If the value of e1 is non-NIL then (OR e1 ... eN) reduces to the reduction of e1.

If e1 has the value NIL then (OR e1 ... eN) reduces to the reduction of (OR e2 ... eN).

If e1 has no value then (OR e1 ... eN) reduces to (OR e1' e2 ... en), e1' being the reduction of e1.

All of this corresponds to LISP usage, the disjuncts being taken in order and only as far as necessary to determine the result.

4.6.6 Conditionals

(COND q1 ... qN)

(COND) reduces to NIL.

If q1 is (e0 ... eM) then:

if e0 has no value then (COND q1 ... qN) reduces to (COND (e0' ... eM) ... qN), where e0' is the reduction of e0;

if e0 has the non-NIL value v, then (COND q1 ... qN) reduces to the reduction of (PROGN (QUOTE v) e1 ... eM) [note that (QUOTE v), rather than simply v, is needed here since it is possible that M = 0];

if e0 has the value NIL then (COND q1 ... qN) reduces to the reduction of (COND q2... qN).

All of this conforms to customary LISP practice, since PROGN mimics the sequential evaluation of the expressions in a conditional "arm".

4.6.7 Sequential Compositions

(PROGN e1 ... eN)

(PROGN) reduces to NIL.

(PROGN e) reduces to the reduction of e.

If e1 has no value then (PROGN e1 ... eN) reduces to the reduction of (PROGN e1' e2 ... eN), e1' being the reduction of e1.

If e1 has a value then (PROGN e1 ... eN) reduces to the reduction of (PROGN e2 ... eN), and the side-effect, if any, of evaluating e1 occurs.

(PROG1 e1 ... eN)

(PROG1) reduces to itself.

(PROG1 e) reduces to the reduction of e.

If e1 has no value then (PROG1 e1 ... eN) reduces to (PROG1 e1' e2 ... eN), e1' being the reduction of e1.

If e1 has the value v then (PROG1 e1 ... eN) reduces to the reduction of (PROGN e2 ... eN (QUOTE v)), and the side-effect, if any, of evaluating e1 occurs.

(PROG loc s1 ... sN)

PROGs are always reduced.

There is no reasonable way to carry out a reduction of a PROG analogous to the reduction of PROG1 or PROGN expressions, and the necessity of assignment to the local identifiers of the PROG would lead to limited utility of such a construct, even if we were to define some notion of reducibility for PROGs. PROG may, of course, be used freely in the definitions of functions invoked from LOGIC.

4.6.8 Assignments

```
(SETQ ident e)
```

If e has the value v and ident is a proper identifier then (SETQ ident e) reduces to v, and assigns v to ident as a side effect. Of course, any other side effect of evaluating e also occurs.

If e has no value then (SETQ ident e) reduces to (SETQ ident e'), where e' is the reduction of e. The assignment side effect does NOT occur.

Note that assignment (and indeed any other side effects) should be used with some caution in LOGIC, since the order in which evaluations are performed is determined in part by the heuristic search methods, and thus is not readily predictable.

Observe too that in order to obtain the LISP value of an identifier ident one must write "(EVAL ident)", not just "ident". That is, "(EVAL ident)" reduces to v (or to (QUOTE v), as the case may be), where v is the value of ident. If ident has no LISP value (that is, is "unbound") the attempt to reduce (EVAL ident) will produce the LISP error message "UNBOUND VARIABLE". Ideally, in this case, (EVAL ident) would simply be returned as its own reduction. However, the present implementation takes the more practical view that such a course would be too costly to justify (the overhead involved in the extra testing being possibly quite considerable).

4.6.9 Selections

```
(SELECTQ e (q1 . s1) ... (qN . sN) u)
```

Here s1, ..., sN are lists of expressions.

The reduction of the SELECTQ expression is basically the same as that of the expression

```
(COND ((OR (MEMQ q1 '(T OTHERWISE)) (EQ e q1)) . s1)
         .
         .
         .
      ((OR (MEMQ qN '(T OTHERWISE)) (EQ e qN)) . sN))
```

except that reductions are expressed with SELECTQ and e is evaluated just once at the beginning. If one of the selection keys qi is a list (i1 ... im) then the corresponding disjunct of the COND predicate is

(MEMQ e (LIST i1 ... im))

4.7 LOGLISP SPECIAL FORMS

The remaining special forms do not occur in conventional LISP. They provide means by which the LOGIC programmer may control the interaction between LOGIC and LISP in order to deal with various unusual circumstances.

These special forms deal with the issues raised by the fact that LISP objects can meaningfully be interpreted not only (1) as LISP programs capable of being (in all cases) reduced and (in many cases) evaluated, but also (2) as LOGIC expressions acting as part or all of a predication or clause.

The LISP programmer is accustomed to this situation. One of LISP's most distinctive features is that all LISP programs are also LISP data objects. The device of quotation permits the LISP programmer to coin a name for any expression e by simply writing: (QUOTE e). The name of this name is (QUOTE (QUOTE e)), and so on.

In LOGLISP we have to deal with the fact that LISP and LOGIC are "mutually embedded" but are organized on rather different semantic principles. LISP is based on the idea of denotation and its main semantic operation is EVAL. LOGIC is based on reduction and substitution (instantiation).

The process of constructing the LISP-reduction of an expression is actually carried out by code written in LISP (although the user need not be aware of this). This code invokes LISP's EVAL and APPLY under suitable safeguards and does its best to provide, in LISP-reducing e, the effects and the outcome that e may call for as a meaningful LISP construct. However, since reduction of e is NOT identical to evaluation of e in every case, the LOGLISP programmer must either stay away from those cases where the notions diverge or else master the differences and the tools we provide for exploiting these.

These tools consist of the following family of "quotation" constructs. Each provides a way of "immunizing" an expression e during the reduction process, in a way similar to the way in which (QUOTE e) "immunizes" e from being evaluated during the evaluation process.

Since these forms do not occur in LISP, it is not already established what, if any, their values are. Hence in the following discussion we shall give in each case not only the reduction but also (where appropriate) the value.

(LOGIC-EXPRESSION e)   [short form: (LOGIC e)]

Intuitively, (LOGIC-EXPRESSION e) specifies that the result of evaluation is to be interpreted as a LOGIC expression rather than as a LISP object. The most obvious effect of this is to suppress the quoting of non-atomic values which would otherwise occur.

If e has the value v, and if v has the value w, then (LOGIC-EXPRESSION e) reduces to w (or to (QUOTE w), as the case may be) and also has w as its value. If v has no value, then (LOGIC-EXPRESSION e) reduces to the reduction of v.

If e has no value (LOGIC-EXPRESSION e) has no value, but reduces to (LOGIC-EXPRESSION e') where e' is the reduction of e.

Put differently, when e has a value v, we reduce (LOGIC-EXPRESSION e) by treating v as a LOGIC expression and reducing v. In practice it often happens that v is reduced, in which case (LOGIC-EXPRESSION e) reduces to v.

(LISP-OBJECT e)   [short form: (LISP e)]

(LISP-OBJECT e) is reduced, but has the value e.

In this respect, (LISP-OBJECT e) is like (QUOTE e). However, (LISP-OBJECT e) differs from (QUOTE e) in that (LISP-OBJECT e) is subject to instantiation, that is, (LISP-OBJECT e)*s is (LISP-OBJECT e*s) for all substitutions s.

Note. Logicians will recognise this as the device of "quasi-quotation" which first appears in W. V. O. Quine's Mathematical Logic (1940). The point of it is that one often needs to consider pieces of text which are "quotation schemas" - i.e., they are just like quotations except that they contain one or more "slots" awaiting further specification. Thus (QUOTE (+ x 2)) names the expression which is a 3-list whose successive elements are the atoms "+", "x", and "2"; whereas (LISP-OBJECT (+ x 2)) is an expression which can become (LISP-OBJECT (+ 3 2)), or (LISP-OBJECT (+ 5 2)), etc., by substitution for "x". End of note.

(QUOTE-ONLY-IF-GROUND e)   [short form: (GROUND e)]

The form (QUOTE-ONLY-IF-GROUND e) is similar to (LISP-OBJECT e), but has a value only if no variables occur in e. More precisely, (QUOTE-ONLY-IF-GROUND e) is reduced, but has a value only if no variable occurs in e, in which case its value is e.

(LOGIC-GR e)

(LOGIC-GR e) is equivalent to (LOGIC (QUOTE-ONLY-IF-GROUND e)). It follows that if any variable occurs in e then (LOGIC-GR e) has no value and is reduced.

If no variable occurs in e then the reduction and value of (LOGIC-GR e) are those of e.

(IRRED e)

(IRRED e) has no value, and its reduction is e (not the reduction of e). This form may be used to suppress reduction of an expression which may not be reduced.

(Variable e)

(Variable e) has value and reduction T if the expression e (instantiated) is a variable, value and reduction NIL otherwise. "Variable" is, in fact, the name of a MACRO defined by

```
(DEFMACRO Variable (X) '(VARIABLE ,X))
```

We shall illustrate a few applications for these forms. First, consider the expression

```
(LOGIC (SUBST (GROUND x) (GROUND y) (GROUND z)))
```

which, as it stands, has no value and is reduced. Suppose now we instantiate it,using the substitution

```
x = (+ (VAR A) 3)
y = (VAR Q)
z = (<= (VAR Q) 10)
```

to obtain the expression

```
(LOGIC (SUBST (GROUND (+ (VAR A) 3))
              (GROUND (VAR Q))
              (GROUND (<= (VAR Q) 10))))
```

where VAR is not the name of a LISP function. Since no variables now occur in the GROUND expressions these now have values, hence so does the expression (SUBST ...), and hence the whole reduces to

```
(<= (+ (VAR A) 3) 10)
```

4.8 SIMPLIFYING IMPLICIT CONSTRAINTS--THE FUNCTION SIMPLIFY

If c = (q env) is an implicit constraint then (SIMPLIFY c) is the implicit constraint which results from reducing one or more of the predications in c and dropping them if they reduce to "true". Specifically, (SIMPLIFY c) is the result of the following three-step algorithm:

```
1  let q be (CAR c) and env be (CADR c)

2  while  q is nonempty

   do     let b*{env} be the reduction of
              (CAR q)*{env}

          if b*{env} is "true"
             then replace q by (CDR q)
             else return (LIST (CONS b (CDR q)) env)

3  return (LIST NIL env)
```

By "true" we mean any expression which has a value that is not NIL.

4.9 THE EXTENDED DEDUCTION CYCLE

In the actual LOGIC cycle of our LOGLISP system we include a step of simplification in step 1 of the RUN loop. The full description of the loop is then:

```
RUN:  while   WAITING is nonempty

      do  1   remove some c from WAITING
              and let (x y) be (SIMPLIFY c)

          2   if   (x y) is solved
              then add (x y)  to SOLVED
              else add the resolvents of (x y) to WAITING
```

Note that the predication resolved away is the one which was just processed by SIMPLIFY and that it is therefore a reduced expression. In particular it may be the expression NIL (i.e. the LISP representation of falsehood). In this case, there will be no resolvents forthcoming and (x y) will therefore be a failure.

4.10 CONTROLLING REDUCTION

It is sometimes helpful to inform LOGIC that an expression is reduced, either because it is known in advance that reduction will merely reproduce the expression itself, or because reduction would for some reason be inappropriate. This can be accomplished by invoking the LISP MACRO IRREDUCIBLE with a command of the form

```
(IRREDUCIBLE id1 ... idn)
```

id1,...,idn being proper identifiers. This having been done, any expression of the form (idk ...) will thereafter be treated as reduced, regardless of the nature of its subexpressions. The effect of IRREDUCIBLE can be undone with

```
(REDUCIBLE id1 ... idn)
```

(REDUCIBLE is also an MACRO). REDUCIBLE will not, however, repeal the system-mandated immunity of PROGs to further reduction.

These matters are discussed further in Chapter 5, Creating Knowledge Bases.

4.11 SUBSCRIPTED VARIABLES

We have mentioned before that the variables occurring in clauses are, in effect, renamed before resolution so as to prevent unintended identification of variables in different clauses. This is accomplished by "subscripting" the variables in the clauses with appropriately chosen non-negative integers. Ordinarily this subscripting is hidden from the user, and is, in fact, performed implicitly and quite economically. Subscripted variables may, however, appear in answers to queries, and are routinely seen when monitoring deductions (see Chapter 10). In such cases, the subscripted variable is an identifier whose print name consists of an ordinary variable suffixed by one or more subscripts, each subscript consisting of a "^" followed by one or more digits. Examples are x^7 and date^3^17. Such variables, generated by the system, are the only variables which may contain "^". User-coined variables may not contain "^".

4.12 UNIFICATION IN LOGLISP

There are a few points worth noting about the LOGLISP implementation of unification.

First of all, there is no check performed to see if a unification has created any cycles. Such a check would, if routinely made, be time-consuming. It appears that in normal LOGIC programming the check is unnecessary. Since unification is confined to the cases where the input expressions do not have variables in common, cycles can arise only if clauses or queries are formulated in certain unusual ways.

The use of implicit representations throughout in any case makes it possible to work with some infinite (cyclic) expressions as though they were finite (which in a suitable sense they are). It is only when a sophisticated user wishes to exclude such expressions from the domain of discourse that their detection becomes necessary.

Of course, any process (such as a naive recursive realization) which seeks to traverse every path in such an expression will run on indefinitely, and the user will want to avoid this situation. In designing LOGLISP we have assumed that any user deliberately creating such expressions will be sophisticated enough to use LISP to protect himself without being lectured at by us. We have further assumed that any user inadvertently creating such expressions will prefer to take the error messages or other indications of his mistake which LISP will provide - in place of the expensive LOGLISP overhead which would be needed to protect him from them.

4.12.1 Proper Names

Two proper names, say a1 and a2, are considered to be unifiable iff (== a1 a2) where == could be defined by the macro

```
  (DEFMACRO == (X Y)
    `(OR (EQL ,X ,Y)
         (EQUAL ,X ,Y)))
```

This produces just the effect one wants, but note that distinct identifiers with the same PNAME are not unifiable (it cannot be the case that both are INTERNed). The integer 1 unifies with the floating-point numeral 1.0, on the other hand, and distinct occurrences of the same floating-point numeral are unifiable.

4.12.2 Special Forms

Expressions in QUOTE, FUNCTION, and F-L are treated specially by the unifier. (QUOTE e1) unifies with (QUOTE e2) if and only if (EQUAL (QUOTE e1) (QUOTE e2)), and similarly for (FUNCTION f1) with (FUNCTION f2). (F-L . e1) unifies with (F-L . e2) if and only if e1 and e2 are the same list.

In addition to these cases, expressions of the form (CONS e1 e2) may unify with expressions (QUOTE (a . d)). In attempting to unify two such expressions any logic variables appearing in (a . d) will be treated as "constants". Let us define q[v] as follows: if v is a proper name then q[v] is v, otherwise q[v] is (QUOTE v). In attempting to unify (CONS e1 e2) with (QUOTE (a . d)) the unifier proceeds by attempting to unify e1 with q[a], then, if successful, unifying e2 with q[d]. Variables in e1 and e2 will be bound to subexpressions of a and d, QUOTEd when appropriate. Some examples will make things clear. The expression

```
(CONS x y)
```

unifies with

```
(QUOTE (A B C))
```

with mgu x = A, y = (QUOTE (B C)). To take a more complicated case,

```
(CONS (CONS F x) (CONS u v))
```

unifies with

```
(QUOTE ((F (A B)) C D))
```

with mgu

```
x = (QUOTE ((A B))), u = C, v = (QUOTE (D)) .
```

Expressions in QUOTE, FUNCTION, and F-L are not otherwise unifiable. It should be remarked that an expression like (F A QUOTE (B)) does not contain a quotation, merely an occurrence of the constant QUOTE.

4.12.3 Variables As Tails

Ordinarily, an expression is either an atom or a list, but one may, in fact, introduce expressions which are composite but not lists. The only useful expressions of this class are those for which repeated CDR's eventually yield a variable, an example being (P (F x) . y). We remark that the definitions of unification and resolution given in chapters 2 and 3 do not actually require that non-atomic expressions be lists.

In a sense, there is really nothing special about a composite expression which is not a list, but such expressions are sufficiently unusual that further discussion may be in order. Expressions of this sort are particularly useful in dealing with operators which take a variable number of arguments. To illustrate, the expression

```
(+ x . y)
```

unifies with

```
(+ u 7)
```

with mgu

```
x = u, y = (7)
```

and also unifies with

```
(+ (F u 3) 7 (G A B))
```

with mgu

```
x = (F u 3), y =(7 (G A B)) .
```

Thus a simple, but still rather flexible, rule for solving equations involving sums may be asserted by

```
(ASSERT (== (+ x . y) z) <- (== x (- z (+ . y)))) .
```

4.12.4 The "Don't Care" Symbol

The identifier [], called the "don't care" symbol, unifies with any expression whatever, but such a unification introduces no bindings. The effect is as though each occurrence of [] were replaced by a new variable not appearing elsewhere in the expressions to be unified, except that the implementation benefits from use of the don't care symbol.

To illustrate, the expression

```
(P [] x [])
```

unifies with

```
(P (F 1) (G A) 7)
```

with mgu

```
x = (G A) .
```

4.13 REDUCTION OF EXPRESSIONS ENDING IN VARIABLES

The reduction of an expression (f e1 ... eN . v) will now be explained. Such an expression has a value if and only if f is the name of a MACRO and the macro expansion has a value.

If f is a proper identifier, but not the name of a MACRO, then the expression has no value but reduces to (f e1' ... eN' . v), where the ei' are the reductions of the ei.

The sequentially evaluated LISP forms, those formed with AND, OR, COND, PROGN, PROG1 and SELECTQ, may also involve variable tails. Reduction proceeds as described before, stopping when a variable tail is encountered. Such expressions may have a value if the "evaluation path" avoids variable tails entirely.

4.14 SPECIAL RULES FOR RESOLUTION

The system "automatically" incorporates a number of special rules applicable to certain predicate symbols. In most cases these rules are just economical implementations of computations that could be achieved with ordinary clauses, but the rule for CONDitional expressions constitutes a fundamental extension of the system, as it introduces a form of "negation as failure".

Application of any of the rules can be enabled or disabled at will by the user.

4.14.1 The Rules

Each of the rules is introduced by an informal, clause-like description, followed by discussion and, in some instances, a nearly equivalent formulation with actual clauses.

4.14.1.1 Equations

```
(== e1 e2) <- "e1 and e2 are unified"
```

The rule is just the reflexive law of equality, and amounts to

```
(ASSERT (== x x)) .
```

4.14.1.2 Conjunctions -

```
(AND p1 ... pN) <- p1 & ... & pN
```

Bearing in mind that (AND) reduces to T, the rule for AND amounts to

```
(ASSERT (AND x . y) <- x & (AND . y)) .
```

4.14.1.3 Disjunctions -

```
(OR p1 ... pN) <- pi, for i = 1 ... N
```

Again, bear in mind that (OR) reduces to NIL. The rule for OR is practically equivalent to the two clauses

```
          (ASSERT (OR x . y) <- x)
     (ASSERT (OR x . y) <- (OR . y))
```

except that resolvents for all of the disjuncts are obtained in one step.

4.14.1.4 Conditionals -

```
(COND (p1 q1) ... (pN qN)) <- pk & qk,
      for the first k such that pk is provable
```

Let us refer to the constraint from which (COND ...) was selected for resolution as the "original constraint". The control mechanism, in fact, begins by attempting to prove p1. If it succeeds in doing so, it introduces a new resolvent consisting of qk and the other predications of the original constraint in the environment which proved p1. (Such a resolvent will eventually be produced for each proof of p1, if the search continues so long.) If all attempts to prove p1 terminate in failure then the control mechanism attempts to prove p2, and so on. All of these searches are carried out within the heuristic limitations imposed on the problem at the beginning. These searches are, moreover, carried out "in parallel" with searches for other solutions to the initial problem, in accordance with the standard heuristic, so that depth-first runaway will be avoided to the extent possible.

The "arms" of the CONDitional expression need not have exactly two expressions. An arm of the form (pk) is, for purposes of resolution, equivalent to (pk T), while an arm of the form (pk qk1 ... qkm) is equivalent to (pk (PROGN qk1 ... qkm)).

This treatment of conditionals depends on a feature of the system not hitherto mentioned, namely the ability to associate a "continuation" with a node. The continuation is itself just a node of a somewhat special nature which is not itself available for computing resolvents. We write a node C with continuation K as "[C Continuation: K]". The resolvents of [C Continuation: K] are exactly the nodes [R Continuation: K] such that R is a resolvent of C.

Let (q env) be a node whose resolvents are desired, let (CAR q) be p, and suppose that p{env} has the form (COND (p1 q1) ... (pN qN)). We obtain a "resolvent" which is

```
  [((p1) env)
   Continuation:
   ((LOGLISP:CONDITIONAL (q1) (p2 q2) ... )*q' env)]
```

where q' is (CDR q).
Each proof of p1 generates a resolvent (NIL envz) with
the same continuation, from which we "pop up" the
continuation to obtain a resolvent (((q1).q') envz).
If and when all attempts to prove p1 fail,
we pop up the continuation to obtain

```
                (((COND (p2 q2) ... (pN qN)).q') env)
```

s 1
which is added to WAITING.

Continuations are not usually printed when explaining
answers or monitoring deductions, rather the fact
that a node has a continuation is indicated by
printing "CONTINUED". Users
can instruct the system to print continuations in
full by invoking the command (CONTINUATIONS ON).
(CONTINUATIONS OFF) returns the system to the normal mode.

4.14.2 Controlling The Special Resolution Rules

All of the rules may be enabled or disabled by invoking functions of the form (AUTO-x "flag") where flag may be either :ON or :OFF. The complete set of control functions for the resolution rules is

```
(AUTO-== "flag")
(AUTO-AND "flag")
(AUTO-OR "flag")
(AUTO-COND "flag")
```

Each macro returns its argument. T or NIL may be used instead of :ON or :OFF. One may also type the nested expression

```
*(AUTO-AND (AUTO-OR :OFF))
```

to disable both the AND rule and the OR rule. All of the rules are enabled by system initialization, hence by RESTORE-LOGIC (see the chapter on filing knowledge bases).[>>>Chapter 8]

CHAPTER 5

CREATING KNOWLEDGE BASES

To create a knowledge base one begins with the empty knowledge base and adds clauses to it one at a time as explained below. Or one can extend an already existing knowledge base by installing it in a LOGLISP workspace and adding more clauses to it. The empty knowledge base is created by executing the command

```
(START)
```

which discards any clauses already present and initializes the LOGIC part of the workspace (without affecting the LISP definitions, if any, which the user may have set up).

5.1 ADDING A CLAUSE TO THE KNOWLEDGE BASE

The <u>assertion</u> <u>command</u>

```
(ASSERT B <- A1 & ... & An)
```

causes the clause B <- A1 & ... & An to be added to the current knowledge base.

The arrow and the ampersands may be omitted. We shall sometimes omit them in the examples in this manual.

5.1.1 Naming A Clause

A clause may be given a user-coined name. This is most conveniently done at the time the clause is added to the knowledge base, using an <u>extended assertion command</u>. Execution of the extended assertion command

```
(ASSERT N B <- A1 & ... & An)
```

adds the clause B <- A1 & ... & An to the current knowledge base, as before, but also ascribes to it the name N. The user-coined name N may be any proper identifier. For example, the following four transactions:

```
(ASSERT (Born Herbrand 12 February 1908))
ASSERTED
```

```
(ASSERT (Died Herbrand 27 July 1931))
ASSERTED

(ASSERT TURING1 (Born Turing 23 June 1912))
ASSERTED

(ASSERT TURING2 (Died Turing 7 June 1954))
ASSERTED
```

add four clauses to the knowledge base, the first two of which are anonymous, and the second two of which have been named respectively TURING1 and TURING2. Note that each assertion transaction is terminated by the message ASSERTED. If the clause is ill-formed the message returned will be ERROR-Ignored, in which case the knowledge base is not altered by the transaction.

The clauses making up a knowledge base are organized into groups called procedures. All clauses in the knowledge base whose conclusions have the same predicate P are grouped together into a procedure which is called "the procedure P". It is thought of, intuitively, as the portion of the knowledge base which is relevant to establishing those facts in the world whose predicate is P.

Assuming that the knowledge base was empty before the above four clauses were added, the contents of the knowledge base now consists of two procedures, each containing two clauses.

By invoking the PRINTFACTS command [see the following Chapter on Displaying Knowledge Bases] the contents of the knowledge base can be displayed, its clauses organised into procedures. Thus:

```
(PRINTFACTS)
;Knowledge Base:

(DEFINE-PROCEDURE Born ()
  ((Born Herbrand 12 February 1908))
  ((TURING1 (Born Turing 23 June 1912))))

(DEFINE-PROCEDURE Died ()
  ((Died Herbrand 27 July 1931))
  ((TURING2 (Died Turing 7 June 1954))))

;End of Knowledge Base.
DONE
```

If one adds a clause with name N to a procedure which already has a clause named N, then the name is removed from the older clause and attached to the new one. A single proper identifier may, however, be used to name as many clauses as one likes, provided no two of these are in the same procedure.

5.2 THE FACTS MODE

A somewhat more convenient way of asserting a succession of clauses is provided by the FACTS mode. By executing the command (FACTS) the user puts the system into the FACTS mode. This is simply a wait-read-assert cycle which expects successive clauses to be typed in. The prompt-message Assert> is printed by the system to signify its readiness to receive the next clause. Thus the four clauses of our example could have been asserted by means of the following excursion through the FACTS mode:

```
(FACTS)
Assert> ((Born Herbrand 12 February 1908))
ASSERTED
Assert> ((Died Herbrand 27 July 1931))
ASSERTED
Assert> (TURING1 (Born Turing 23 June 1912))
ASSERTED
Assert> (TURING2 (Died Turing 7 June 1954))
ASSERTED
Assert> END-KEY
DONE
```

Such a FACTS session is terminated by hitting the blue END key in response to the Assert> prompt. It should be noted that the format in which a clause B <- A1 & ... & An is typed for input to the FACTS mode is the list (B A1 ... An) . The first item on this list may be the optional user-coined name, as illustrated above. The list format enables the system to accept inputs which are too large to fit all on one line. As in the standard LISP convention, the system reads line after line of typed input until a syntactically complete object has been formed. Thus in the following FACTS transaction the three-component clause AGE-FORMULA is asserted on several lines, each of which after the first is prompted by a colon:

```
(FACTS)
Assert> (AGE-FORMULA
                (Age person given-year a)
                (Born person [] [] birth-year)
                (== a (- given-year birth-year)))
ASSERTED
Assert> END-KEY
DONE
```

The clause AGE-FORMULA is now installed as the sole component of a procedure Age which computes a person's age in a given year by looking up the year in which that person was born and subtracting it from the given year. Note the use of the don't care symbol ([]) to match the day and month of birth, neither of which is needed for the deduction. The contents of the knowledge base may again be viewed by executing (PRINTFACTS):

```
(PRINTFACTS)
;Knowledge Base:

(DEFINE-PROCEDURE Born ()
  ((Born Herbrand 12 February 1908))
  ((TURING1 (Born Turing 23 June 1912))))

(DEFINE-PROCEDURE Died ()
  ((Died Herbrand 27 July 1931))
  ((TURING2 (Died Turing 7 June 1954))))

(DEFINE-PROCEDURE Age ()
  (AGE-FORMULA (Age person given-year a) <-
               (Born person [] [] birth-year) &
               (== a (- given-year birth-year))))

;End of Knowledge Base.
DONE
```

The "<-" and "&" appearing in AGE-FORMULA are simply "syntactic sugar" intended to assist the reader in perusing complex clauses. These may also be typed in clauses given to ASSERT or FACTS, but we usually don't bother to do so.

An ill-formed clause typed to FACTS will be ignored, and a message will be typed to inform the user. This HELP message will also be typed in response to the user hitting the blue HELP key following the Assert> prompt.

5.3 ADDING CLAUSES FROM LISP FUNCTIONS

The assertion function ASSERT is just a LISP MACRO, and as such may be invoked by any LISP function. LISP programmers will usually find it more convenient, however, to use the function ASSERT* of one argument, whose value should be a list as might be typed to FACTS (or appear as the tail of an invocation of ASSERT). If the clause is well-formed it will be added to the knowledge base and ASSERT* will return NIL. If the clause is ill-formed it is ignored and ASSERT* returns ERROR.

5.4 ORDER OF CLAUSES IN THE KNOWLEDGE BASE

The order of the clauses within a single procedure is first the data, if any, in the order in which they were asserted, then the rules of the procedure, in the order in which they were asserted. This is the order in which the clauses are printed by PRINTFACTS.

The order of the procedures in the knowledge base is the order in which clauses for the procedures were first asserted. This also is the order used by PRINTFACTS. It should be noted that the order of procedures is frequently changed by editing (see Chapter 7).

5.5 DECLARING ATTRIBUTES OF PROPER IDENTIFIERS

One may ascribe various attributes to proper identifiers in order to influence the operation of LOGIC. An example is :IRRED , the attribute which indicates irreducibility, and others will be introduced later. Several methods are provided for declaring such attributes.

```
(PROCEDURE "id" "at1" ... "atn")        [MACRO]
(CONSTANT  "id" "at1" ... "atn")        [MACRO]
```

Either of these sets the attributes of the proper identifier id to (at1...atn), having first erased any previous attributes. Thus (PROCEDURE ID) declares that ID has no special properties. PROCEDURE is intended for use with predicates, CONSTANT for use with other identifiers, but both names in fact invoke the same function. PRINTFACTS displays attributes of predicates in a list following the predicate name in the DEFINE-PROCEDURE statement. The ()s following Born, Died, and Age in the above example are empty attribute lists.

(ADD-DECLARATION "atr" "id1" ... "idn) [MACRO]

adds attribute atr to those already declared for identifiers id1,...,idn.

(REMOVE-DECLARATION "atr" "id1" ... "idn") [MACRO]

removes attribute atr from among those presently declared for identifiers id1,...,idn.

As mentioned earlier, alternative means are provided for declaring identifiers irreducible.

(IRREDUCIBLE "id1" ... "idn") [MACRO]

declares id1,...,idn to be irreducible (attribute :IRRED), retaining any previous attributes.

(REDUCIBLE "id1" ... "idn") [MACRO]

erases the attribute :IRRED from id1,...,idn, without affecting other attributes.

(IRREDUCIBLE* L) [FUNCTION]
(REDUCIBLE* L) [FUNCTION]

The argument L should be a list of proper identifiers. Each function has the same effect as the corresponding MACRO, for the identifiers listed.

One may also declare attributes of identifiers while in FACTS mode. To do so, one types a line of the form

```
Assert> (id at1 ... atn)
DECLARED
```

in response to the prompt "Assert> ". The effect is to declare at1,...,atn as attributes of id <u>in addition</u> to any previous attributes. Just as one can enter assertions over many lines, so one can type such declarations over many lines if it should ever seem necessary.

The attributes used by LOGIC are :IRRED, :ONERES , :HIST and (:INDEX . ixl). :IRRED has already been explained. :ONERES and (:INDEX ...) will be discussed in Chapter 9, while :HIST is treated in Chapter 11. Other attributes may be declared and will be recorded, but have no effect on the operation of the system.

A short sample session with LOGLISP:

```
(START)
DONE
(FACTS)
Assert> ((Occupation Herbrand Mathematician))
ASSERTED
Assert> ((Occupation Turing Mathematician))
ASSERTED
Assert> (Occupation :HIST :ONERES)
DECLARED
Assert> END-KEY
DONE
(PRINTFACTS)
;Knowledge Base:

(DEFINE-PROCEDURE Occupation (HIST ONERES)
  ((Occupation Herbrand Mathematician))
  ((Occupation Turing Mathematician)))

;End of Knowledge Base.
DONE
```

5.6 ADDING PROCEDURES VIA DEFINE-PROCEDURE

DEFINE-PROCEDURE is a built in LOGIC macro and as such allows the user a fourth method (others are ASSERT, ASSERT*, and FACTS) for entering assertions into the knowledge base. The LISP expression below

```
(DEFINE-PROCEDURE p (at1 ... atN) asrn1 ... asrnM)
```

macro expands to the following LISP expression

```
(PROGN
   (ERASEP p)
   (PROCEDURE p at1 ... atN)
   (ASSERT* (QUOTE asrn1))
   .
   .
   .
   (ASSERT* (QUOTE asrnM)))
```

which first erases the entire procedure p (if it existed) from the knowledge base, then assigns attributes at1 ... atN to p, and finally adds assertions asrn1 ... asrnM to the knowledge base. The user may enter procedures into the knowledge base by typing at a Lisp Listener DEFINE-PROCEDURE macros or, more conveniently, create, in an editor buffer a collection of macro calls. These procedures, entered into the editor buffer, can then be installed by evaluating the buffer (via the extended editor command META-X "evaluate buffer"). After a procedure p has been entered into the knowledge base in this manner one can edit it by going into the editor (using any convenient method of entrance) and executing the "edit definition" command (META-. p) and reinstall it by reevaluating the edited expression (via HYPER-CONTROL-E for example).

5.7 CONVENTIONS FOR DISTINGUISHING VARIABLES

The normal convention is that symbols beginning with lower case letters are LOGIC variables, and that all other symbols are proper identifiers. Other conventions can, however, be adopted.

```
(VARIABLES "vs")                              [MACRO]
(VARIABLES* vs)                               [FUNCTION]
```

set the variable convention according to vs and return the former convention. If vs is NIL the convention is not changed, and the current convention is simply returned. Besides NIL, allowed values for vs are

1. The atom LC to specify the (default) lower case convention

2. The atom UC to specify that identifiers beginning with upper case letters are variables

3. The ASCII code for a character which will begin all variables

4. A single character identifier giving the initial character for variables

To illustrate, starting with LOGLISP freshly loaded,

```
(VARIABLES NIL)
LC

(VARIABLES ?)
LC

(VARIABLES NIL)
?

(ASSERT (Member ?x (?x . ?ls)))
ASSERTED
```

```
(ASSERT (Member ?x ([] . ?ls)) <- (Member ?x ?ls))
ASSERTED
```

defines a membership relation on expressions akin to, but not at all the same as, MEMBER for lists, using the new convention.

It is not intended that one mix variable conventions within a knowledge base, though it is actually possible to do so in some situations. The determination that an identifier is or is not a variable is made at the time the identifier enters the LOGIC part of the system, as when a clause is entered or a query submitted, and subsequent changes in the convention cannot alter that determination.

5.8 CONVERTING VARIABLES TO OTHER CONVENTIONS

Since the programmer may choose from a number of conventions for distinguishing variables from identifiers, it is sometimes desirable to assert clauses written with different conventions into the same knowledge base, particularly when the clauses in question have been recorded in files on disk. We consequently provide means for converting variables from one convention to another, so that the resulting knowledge base will exhibit a uniform convention for naming variables.

To accomplish this, we allow the user to establish two conventions for distinguishing variables from proper identifiers, an "input" convention which will be used to recognize variables in expressions submitted to LOGIC, and an "output" convention in which these variables will be represented in the knowledge base.

```
(CONVARIABLES "vs")                    [MACRO]
(CONVARIABLES* vs)                     [FUNCTION]
```

establish the input convention according to vs (specified as for VARIABLES), leaving the previous convention as the output convention, and return the previous, now output, convention. If vs is NIL the input convention is set to the output convention and conversion is disabled. Variables are converted to the output convention by prefixing a single character to the print name: "v" if the output convention is LC, "V" if it is UC, and the character which distinguishes variables in any other convention.

When conversion has been enabled by invoking CONVARIABLES the input convention can be changed using either CONVARIABLES or VARIABLES. In this mode of operation VARIABLES reports the previous input mode. The output convention cannot be changed until conversion has been disabled by (CONVARIABLES NIL).

Note that print names of proper identifiers are never altered, even if these would be treated as variables in the output convention. If such identifiers occur they will be treated as proper identifiers in the knowledge base, but some confusion is possible when clauses are printed, or if such identifiers are extracted and later re-entered into LOGIC using LISP.

5.9 SUBSCRIPTED VARIABLES IN CLAUSES

Although it rarely happens in practice, one might attempt to assert a clause containing subscripted variables. For technical reasons, subscripted variables may not appear in the knowledge base. If one does attempt to assert a clause containing subscripted variables, or variables in the sequence genvar1, genvar2, ..., the system will rename such variables, using variables genvar<numeral>, so that the clause which results in the knowledge base is a variant of the assertion which was entered, and has no subscripted variables. When a non-standard variable convention is in effect the generated variables are adjusted appropriately.

CHAPTER 6

DISPLAYING KNOWLEDGE BASES

Various commands are provided for viewing the contents of a knowledge base.

6.1 DISPLAYING THE ENTIRE CONTENTS OF A KNOWLEDGE BASE

The command (PRINTFACTS) causes the system to print out a display of the entire current knowledge base.

The display is organised into groups of clauses preceded by the message ";Knowledge Base:". Each group of clauses constitutes a (logical) procedure. That is to say, the header of every clause in the group has the same predicate (say, P). A procedure P, having attributes AT1,...,ATn, naming the collection of clauses C1,...,Ck, is displayed in the following way:

```
(DEFINE-PROCEDURE P (AT1 ... ATn)
  C1
   .
   .
   .
  Ck)
```

where each Ci, having head A and body B1,...,Bm is displayed:

```
    (A <-
     B1 &
       .
       .
       . &
     Bm)
```

The order in which the clauses appear in the display is data first, then rules, in the order in which they were asserted within each class. The display is terminated by the message ";End of Knowledge Base.".

6.2 DISPLAYING A PROCEDURE

The command (PRINTFACTSOF P) displays the procedure P in the same style as that of the (PRINTFACTS) display. If one wishes to print several procedures P1, ..., PN one types (PRINTFACTSOF P1 ... PN).

The command (PRLENGTH P) returns the number of assertions in the procedure P:

```
(PRLENGTH Born)
2.
```

6.3 DISPLAYING THE SET OF DEFINED PREDICATES

The command (PREDICATES) returns a list of the predicates for which logic procedures are defined in the current knowledge base. With the example of the preceding chapter we have:

```
(PREDICATES)
(Born Died Age)
```

The command (CONSTANTS) returns a list of the constants which have been declared. These are proper identifiers other than predicates which have special LOGIC attributes.

6.4 DISPLAYING DATA IN WHICH A GIVEN PROPER IDENTIFIER OCCURS

It is often convenient to be able to retrieve and display the set of data in a given knowledge base in which a given notion occurs explicitly. Such a set in some sense corresponds to what the knowledge base says about that notion in a direct way. The command (PRINTCREFSOF C) displays all data in which the constant C appears somewhere. These clauses are organized into groups by their procedure name, but the entire procedure is not necessarily shown (only those of its data are shown which actually contain C).

Given that:

```
(CONSTANTS)
(Herbrand Turing)
```

then

```
(PRINTCREFSOF Turing)
Turing

(TURING1 (Born Turing 23 June 1912))

(TURING2 (Died Turing 7 June 1954))

Turing
```

6.5 RETRIEVING A PROCEDURE AS A LIST

The procedure P may be obtained as a LISP data object, namely, as the list of its constituent clauses. This list is returned as the value of the command

(ASSERTIONSOF P)

Each clause B <- A1 & ... &An in the procedure is represented as the list (B A1 ... An). If the clause has the user-coined name N then it is represented as the list (N B A1 ... An). For example, (ASSERTIONSOF Born) returns the list

```
(((Born Herbrand 12. February 1908.))
 (TURING1 (Born Turing 23. June 1912.)))
```

The result of ASSERTIONSOF shares no list structure with the internal representation of the knowledge base, thus list-altering operations such as RPLACA and RPLACD performed on this list will have no effect on the knowledge base.

6.6 RETRIEVING INDIVIDUAL CLAUSES

One may display one or more individual clauses using a command of the form

```
(PRINTNA dsg1 ... dsgn)            [MACRO]
```

where dsg1,...,dsgn are "clause designators".

In its simplest form a clause designator is just a clause name, but more elaborate forms may be used to resolve possible ambiguities, and indeed to designate any clause in the knowledge base, whether named or not.

The possible forms for clause designators are shown below. Here 'pred' denotes a predicate, 'name' a clause name, and 'numb' a positive integer.

```
     name                          (possibly ambiguous)
     (pred name)
     (pred numb)                   (possibly ambiguous)
     (pred :DATUM name)
     (pred :RULE name)
     (pred :DATUM numb)
     (pred :RULE numb)
```

As indicated, some of these forms may be ambiguous, depending on the state of the knowledge base. Where a number is given, it specifies the ordinal position of the clause within its class (rules or data) in the indicated procedure. The concise form (pred numb) is ambiguous if the procedure for 'pred' has both a datum 'numb' and a rule 'numb'. The forms (pred :DATUM name) and (pred :RULE name) are redundant, and either is treated as though it were (pred name).

PRINTNA prints the indicated clauses and returns the list (dsg1...dsgn).

An appropriate error message will be printed for any designator which is either ambiguous or fails to designate an clause.

For example:

```
(PRINTNA AGE-FORMULA)
(AGE-FORMULA (Age person given-year a) <-
             (Born person [] [] birth-year) &
             (== a (- given-year birth-year)))
(AGE-FORMULA)

(PRINTNA (Born 1) (Born 2))
((Born Herbarnd 12 February 1908))
(TURING1 (Born Turing 23 June 1912))
((Born 1) (Born 2))

(PRINTNA (Born 3))
No assertion.
((Born 3))
```

One may also retrieve an individual clause as a list. The function

```
(ASSERTION dsg)                         [FUNCTION]
```

returns a list representing the clause designated by (the value of) its argument, if there is one, NIL if the argument fails to designate a clause. Clauses are represenued in the same manner as with ASSERTIONSOF.

6.7 PRINTING CLAUSE NUMBERS

The numbers used to designate anonymous clauses do not ordinarily appear when these clauses are printed, whereas names of clauses do appear. If one wants the numbers to be printed as well, the (LISP) variable *ASRNNUMBERS should be set to any non-NIL value.

To illustrate with a small example:

```
(PRINTFACTSOF Older)

(DEFINE-PROCEDURE Older()
   ((Older Drobny Rosewall) <-)
   ((Older Rosewall Goolagong) <-)
   ((Older x z) <-
    (Older x y) &
    (Older y z))
   ((Older x y) <-
    (Before x y)))

(Older)

(SETQ *ASRNNUMBERS T)
T

(PRINTFACTSOF Older)

(DEFINE-PROCEDURE Older()
   (1 (Older Drobny Rosewall) <-)
   (2 (Older Rosewall Goolagong) <-)
   (1 (Older x z) <-
    (Older x y) &
    (Older y z))
   (2 (Older x y) <-
    (Before x y)))

(Older)
```

Note that data and rules are numbered separately. It should, however, be easy to distinguish the two. It is perhaps worth pointing out that one can properly install a clause by typing

```
(ASSERT 2 (Older x y) <- (Before x y))
```

but that the integer "2" is treated as sugar, and consequently has no effect on the position of the clause in the procedure.

When *ASRNNUMBERS is non-NIL, ASSERTIONSOF and ASSERTION include numbers in the lists they return.

CHAPTER 7

EDITING KNOWLEDGE BASES

The most convenient place (and in the current implementation the only place) to edit clauses is in Zmacs, the Lisp editor. To do this, create the knowledge base in an editor buffer using DEFINE-PROCEDURE, and install it via the extended editor command META-X "evaluate buffer". To then edit a procedure, simply reenter the editor, edit the DEFINE-PROCEDURE form and reevaluate it (one way would be HYPER-CONTROL-E). This method of interaction allows the user to make use of the extensive editing capabilities of Zmacs.

7.1 REMOVING PROCEDURES FROM THE KNOWLEDGE BASE

If one wishes to remove one or more procedures P1, ..., PN from the current knowledge base one invokes the command (ERASEP P1 ... PN).

7.2 DELETING CLAUSES

A number of special functions are provided for deleting selected clauses from the knowledge base. In most cases we provide both macros for use from the terminal, and functions intended to be called from LISP functions.

```
(DELETEN "dsg1" ... "dsgn")          [MACRO]
```

deletes the clauses designated by dsg1,...,dsgn.

Inappropriate designators are ignored, and DELETEN returns a list of designators for clauses which were actually deleted.

```
(DELETENM dsg)                       [FUNCTION]
```

deletes the clause designated by dsg, if there is one. DELETENM returns T if a clause was deleted, NIL otherwise.

```
(DELETE= . "cls")                    [MACRO]
(DELETE=* cls)                       [FUNCTION]
```

Each of these functions deletes the clause which is EQUAL to the specified clause, if there is one. Clause names and "sugar" in cls are ignored in determining equality. Either function returns

T if the specified clause was found and deleted, NIL otherwise.

The following examples illustrate the use of DELETE= and DELETE=* in the context of the example used earlier:

```
(DELETE= (Born Turing 2 June 1912))
T
(DELETE=* '((Died Turing 23 October 1954)))
T
```

The effect of these is to delete the two clauses giving dates of birth and death for Turing. Note that when using these to delete rules the variables specified in the parameter to DELETE= or DELETE=* must be the same as those appearing in the knowledge base.

```
(DELETEA . "cls")                  [MACRO]
(DELETEA* cls)                     [FUNCTION]
```

The argument specifies a clause, as with DELETE= and DELETE=*. All clauses which are instances of the specified clause are deleted. Either function returns T if at least one clause was deleted, NIL otherwise. The predicate of the header of the argument must be a proper identifier, not a variable.

```
(DELETER . "cls")                  [MACRO]
(DELETER* cls)                     [FUNCTION]
```

These functions are like DELETEA and DELETEA*, except that only rules will be deleted.

```
(DELETED . "cls")                  [MACRO]
(DELETED* cls)                     [FUNCTION]
```

The same, except that data are deleted.

CHAPTER 8

FILING KNOWLEDGE BASES

The current knowledge base may be preserved in a file by the LOGIC primitive SAVE-LOGIC.

```
(SAVE-LOGIC pathname &OPTIONAL (verify T) (compile T) (package
:USER))
```

The function SAVE-LOGIC writes to the file (specified by pathname) the current knowledge base (using the function PRINTFACTS) preceded by the two lines displayed below:

```
     ;;; -*- MODE: LISP; BASE: 10.; PACKAGE: package -*-

     (VARIABLES conv)
```

The first line is the mode line. The second line is a call on the macro VARIABLES with argument conv (the variable convention in force at the time of the save). SAVE-LOGIC returns the value DONE.

If the argument verify of SAVE-LOGIC is T (the default), then the user is asked to confirm the destination of the save. The reason for this optional verification step is that the SAVE-LOGIC function "completes" any ambiguous pathname that the user supplies. If this completed pathname is not what the user wished, the verification step allows him to provide a complete pathname himself.

If the compile argument has value T (the default), then, in addition to the base being saved in PRINTFACTS format, the file is compiled and the compiled version is saved (the file having a .QFASL extension). A compiled knowledge base can be loaded with the same commands that load a "source" file. The advantage of compiling is that compiled knowledge bases load two to three times faster than "source" knowledge bases.

An example of a call on the SAVE-LOGIC function.

```
(SAVE-LOGIC "D1;KB")
OK to use: #<FS:LM-PATHNAME LAM1: D1; KB.LISP#> ># (Y or N)? Y
Saving ...
Compiling ...
DONE
```

8.1 RESTORE-LOGIC AND LOAD-LOGIC

A knowledge base that has been saved can be reinstalled using either of the two functions

```
(RESTORE-LOGIC pathname &OPTIONAL (verify T) (package :USER))
```

or

```
(LOAD-LOGIC pathname &OPTIONAL (verify T) (package :USER)).
```

A call on LOAD-LOGIC adds the procedures defined in the file to the existing knowledge base. The function RESTORE-LOGIC, however, first removes all existing assertions from the knowledge base and then installs the file's procedures.

An example:

```
(RESTORE-LOGIC "D1;KB")
Clearing ...
OK to use: #<FS:LM-PATHNAME LAM1: D1; KB.LISP#> ># (Y or N)? Y
Loading ...
DONE
```

Since files created by SAVE-LOGIC are just collections of calls on the macro DEFINE-PROCEDURE, these files may also be loaded using the LISP primitive LOAD. Further, LISP definitions (i.e. DEFUNs, DEFVARs, and DEFCONSTs) may be added to files defining knowledge bases allowing for actual LOGLISP (LOGIC + LISP) files. These files may be installed using any of the three primitives: RESTORE-LOGIC, LOAD-LOGIC, or LOAD.

CHAPTER 9

DEDUCING ANSWERS TO QUERIES

Our informal notion of a query is that it is a description of a set, in the style: the set of all X such that C. We think of the process of evaluating such a description as one of deducing all the different instances X*s of the "answer template" X for which the condition C*s is true.

This type of query is formalized by the LOGIC primitives SETOF and ALL.

For convenience we have also implemented two other query primitives: ANY and THE. ANY intuitively selects, from the set described, a subset of one or more of its elements (which particular ones are selected is left undetermined). THE selects an element of the set (which particular one is selected is left undetermined).

Thus, we might ask for: any 3 members of the set of all X such that C, or: ANY 1 member of the set of all X such that C. In the latter case, the primitive ANY delivers a singleton set. If we want the member of that set, rather than the set itself, we ask for: THE X such that C - just as if there were one and only one such element. The primitive THE does not test for such "existence and uniqueness", however. If no instances of X can be deduced to satisfy C then the "ANY 1" construct returns the empty set while the "THE" construct returns the message "No-solutions-found". "ANY 1" does not care if more than one instance exists, nor should it. "THE" does not care either - as it perhaps should, according to the way the ordinary understanding of the phrase "the ... such that ---" works. We have preferred to leave the uniqueness issue to the user on the grounds that to test routinely for non-uniqueness would cost too much.

In the formal treatment of queries sets are represented by lists. The user can choose (as explained below) whether these are to be construed strictly as sets (with the overhead cost of patrolling for and eliminating duplicate elements) or merely as lists (with possible repetitions).

9.1 ANY, ALL, THE AND SETOF

The deduction machinery of LOGIC is invoked by the deduction commands: ALL, ANY, THE, and SETOF .

The first three are LISP MACROs which may conveniently be invoked from the terminal or within assertions. SETOF is a function intended for use by LISP programs.

9.2 ALL

The command (ALL X C1 ... Cn) returns a list of reductions of the instances of the answer template X with respect to all of the environments which satisfy the constraint (C1 ... Cn) in the current knowledge base. [These environments are called the solutions of the constraint (C1 ... Cn).] If two or more solutions yield the same answer (in the sense that the answer expressions are EQUAL) then the list contains just one instance of the answer, corresponding to the solution obtained first.

The answer template X may be a variable, an atom not a variable, or a list of expressions. We emphasize that the answers returned are the expressions (or lists of expressions) obtained by reducing the instances of the answer template in the solution environments, NOT the values of those expressions. The expressions need not, after all, be evaluable.

9.3 ANY

The command (ANY K X C1 ... Cn) behaves in a similar manner, except that no more than K (distinct) instances of X are returned from among those which the corresponding ALL command would return. K is expected to be a nonnegative integer.

9.4 THE

The command (THE X C1 ... Cn) returns the sole member of the list (ANY 1 X C1 ... Cn) , if there is one, and is intended for use only in contexts where it is known that exactly one solution exists. If no solution exists for the given constraint, THE returns the identifier No-solutions-found.

9.5 SPECIFYING THE DEDUCTION WINDOW

The constraints appearing in invocations of ALL, ANY and THE need not consist entirely of predications. They may also contain control specifications, which affect the nature of the search and treatment of answers, and limit specifications which determine the deduction window to be used. The form of a limit specification is

:Limit Value

where ":Limit" is one of :TREESIZE, :NODESIZE, :ASSERTIONS, :RULES, or :DATA and "Value" is a number, the identifier :INF (denoting infinity) or a non-atomic expression whose LISP value is a number or :INF. Note that the first character of the keyword is ":" -- all keywords are in the user package. It is most often the case that a user is working in the user package. In these situations users may omit the ":"s in front of the keywords. These values determine bounds for the corresponding parameters of the search window. Thus one might, in the context of the "tennis" example of Chapter 3, ask for

```
(ALL x (Male x) (Champion x) (Older x Kelly) :RULES 4)
```

to obtain the set of all those who can be deduced to be male champions older than Kelly with no more than four applications of rules.

In the absence of any specification the limits are all taken to be :INF, except for :RULES, which is never allowed to exceed a limit determined by the implementation, normally 1500.

9.6 SETOF

The preceding commands are special adaptations of the basic general deduction primitive, SETOF.

SETOF takes three arguments. In the command (SETOF S X C) the arguments S, X and C are (LISP) evaluated before the SETOF procedure is entered (SETOF is a function).

The first argument S (the "scope indicator") is an expression which evaluates either to a nonnegative integer or else to the identifier :ALL.

The second argument X is an expression which evaluates to an answer template.

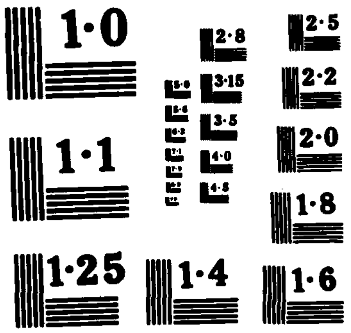
The third argument C is an expression which evaluates to a constraint.

The command (SETOF S X C) returns a list of the recursive realizations of the answer template [which is the value of] X corresponding to the solutions which satisfy the constraint [which is the value of] C in the current knowledge base. If the value of S is :ALL, then all such recursive realizations are in

AD-A158 011 THE LOGLISP PROGRAMMING SYSTEM(U) SYRACUSE UNIV NY 2/2
J A ROBINSON ET AL. MAY 85 RADC-TR-85-89
F30602-81-C-0024

UNCLASSIFIED F/G 9/2 NL

END
FILMED
DTIC



NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

the list returned. If the value of S is the integer K, then no more than K such recursive realizations are returned. Thus the command (ALL (x y) (Age x 1928 y)) is equivalent to the command

```
(SETOF (QUOTE :ALL) (QUOTE (x y)) (QUOTE (Age x 1928 y)))
```

and both return the list

```
((Turing 16) (Herbrand 20))
```

as their result, if the current knowledge base contains only the assertions HERBRAND1, HERBRAND2, TURING1, TURING2 and AGE-RULE. The command

```
(THE logician (Born logician something February 1908))
```

returns the result: Herbrand .

Recall that the answer template may be a proper name, a variable, or a list of expressions. In the first case the answer is (with one exception, explained below) just the answer template. If the template is a variable, each answer is the reduction of the recursive realization of the answer template in a solution environment. If the template is a list of expressions, the answer is the list of reductions of recursive realizations of expressions in the template.

The exceptional answer template is the integer 0. If the template is 0, SETOF (or any of ALL, ANY, THE) returns the number of solutions (not answers) obtained during the search. If the invocation of SETOF limits the number of answers, this limit is taken as a bound on the number of solutions to be found.

9.7 NONDETERMINACY OF DEDUCTIVE PROCESSES

The order of the items in the lists returned by ALL, ANY and SETOF is not defined, nor is there defined any rule for selecting a subset of all instances when less than all are requested.

This non-determinacy is accompanied by a measure of "concurrency", in that the order in which LISP evaluations will be performed in the course of various simplifications is also not specified. The evaluation of a single evaluable expression is, however, carried out "indivisibly". It is for this reason that assignment and other side-effect-producing operations must be used with caution in LOGIC.

9.8 CONTROLLING THE DEDUCTION PROCESS

Having emphasized the non-determinacy of the deduction process, we should now point out that the user can, in fact, exercise a considerable degree of control over it, even to the point of making it fully deterministic.

The search conducted by SETOF is a heuristically guided search, each separate step of which is itself a limited depth-first search, implemented by a backtracking algorithm. Recall that the estimated solution cost of a node (q env) is computed as.

ASSERTIONS(q env) + NODESIZE(q env)

This cost estimate is used both to guide the heuristic search and to limit the depth-first search embedded therein. This works as follows.

For each search a cost increment S is specified, usually by default. The depth-first search is initiated by selecting a waiting node of minimum cost, say C. The bactracking routine then explores the deduction tree starting from the selected node, recording for later consideration any non-terminal node whose cost is as large as, or larger than, C + S. When this limited depth-first search is completed a new "waiting" node of minimum cost is selected to begin another round of depth-first search, and so on.

9.8.1 Search Control

To specify the cost step (i.e. increment) for a particular search, one includes a control specification ":CSTEP s" among the constraints, where s is a positive integer or :INF. Thus (ANY 2 x (P x 7) (Q x) :CSTEP 6) searches with a cost step of 6. When no step is specified a default value, depending on the nature of the search, is used. Searches which seek all answers use the value of the LISP identifier *ALLSTEP, initially 64, as the default. Other searches use the value of *CSTEP, initially 4. We take the view that "ALL" searches should be conducted primarily depth-first, without going so deeply as to run the risk of stack overflow. Searches which may stop with fewer than all solutions are conducted so as to obtain the less costly solutions first.

A pure depth-first search may be obtained as in (ALL x (P x 3) (Q x) :CSTEP :INF), or, of course, by adjusting the default values appropriately.

9.8.2 Answer Control. Lists Versus Sets

Normally, answers are reduced as explained earlier, and answers which duplicate earlier answers (in the sense of EQUAL) are ignored. To suppress reduction of answers include the control symbol :ANS-IRRED among the constraints. To require reduction incorporate the control symbol :ANS-REDUCE. The default control is given by the value of the LISP identifier *REDUCEANS, initially T, indicating that reduction should be performed. The value NIL indicates that reduction should not be performed.

To disable the check for duplicate answers include the control symbol :LIST among the constraints. To require the check incorporate the control symbol :SET. The default control is given by the value of the LISP identifier *SET, initially T to specify that the check for duplicates SHOULD be performed. The value NIL indicates that the check should NOT be performed.

9.9 "ONE RESOLVENT" PROCEDURES

It sometimes happens that the programmer can determine that on every call of a particular procedure at most one resolvent can lead to success. Such a determination usually depends both on the nature of the queries that can be expected and on the nature of the clauses which constitute the procedure. If it can further be arranged that this resolvent always results from the first assertion which yields a resolvent, then one may wish to inform the system of these facts by declaring the procedure in question to have the attribute :ONERES. This is done with the command (PROCEDURE Pred :ONERES), "Pred" being the predicate of the procedure. If a special rule (see Chapter 4) is in effect for "Pred", the special rule is considered to come between data and rules.

The conditions under which one may appropriately choose to specify a procedure to be :ONERES may seem rather restrictive, but they are not uncommon in practice. An inappropriate :ONERES attribution will, of course, have a drastic effect on the meaning of a procedure, since the system will indeed compute at most one resolvent for each call, even if more than one resolvent can lead to success.

9.10 INDEXING CLAUSES FOR QUICK RETRIEVAL

We mentioned in an earlier chapter that data are automatically indexed according to the proper identifiers which occur in them so that LOGLISP can quickly obtain a (usually small) list of candidates for resolution with a given predication. In fact, the indexing scheme takes account of FIXNUMs as well, which are hashed into a convenient number of equivalence classes, normally 47. For technical reasons, this indexing ignores quotations, whether formed with QUOTE, FUNCTION, or F-L, and expressions of the form (CONS ...), since the latter may unify with quotations. The atom NIL, which occurs very frequently, is also ignored.

The actual indexing method is extremely simple. Associated with each proper identifier id and each predicate symbol pr in whose data id occurs, we maintain a list of the data of pr containing id, along with a count giving the number of data in the list. When asked to obtain resolvents for a predication q headed by the predicate pr, the system scans q for proper identifiers, and attempts unification with only those data in the shortest of the associated lists, examining the entire collection of data for pr in the case that q has no proper identifiers.

9.10.1 Indexing Rules

We provide a rather different scheme for indexing rules, which is invoked only at the direction of the programmer. Rule indexing for a predicate pr is specified by declaring pr to have an attribute of the form (:INDEX k1 ... kN), where the k's are positive integers and k1 $\geq$ 2. A predicate can have at most one :INDEX attribute, declaring a new one deletes the old. The integers k1,...,kN define a path into predications headed by pr, namely, of the k1-th entry, the k2-th entry, of which the k3-th entry, and so on. To illustrate, if we specify (:INDEX 2 1) for the predicate "Term", then we select from (Term (F x y) ...) the identifier F. To be effective, the path should be chosen so that the rules of pr have identifiers at the specified position in their conclusions, identifiers which will serve to classify the rules into a number of (preferably small) subsets. It is not, however, mandatory that the path be so chosen, and in fact the path need not even be defined for all rules of pr. An :INDEX declaration can never affect the results of a LOGIC computation, only the performance. An inappropriate :INDEX specification may, however, be just slightly worse than no indexing at all.

When a predication headed by pr is selected for resolution the system examines the component of the predication at the location specified by the :INDEX attribute. If this component is defined and is a proper identifier or FIXNUM, then the corresponding rules are used as candidates for resolution. In any other case LOGLISP will still determine a suitable set of candidates, possibly all of the rules for the predicate pr.

9.11 SUBSCRIPTED VARIABLES IN DEDUCTIONS

We have already mentioned that variables appearing in clauses are (implicitly) given subscripts when the clauses are used in deductions, so as to avoid improper identification of variables.

Variables in the query are given the subscript 0.

For an unsubscripted variable, say x, the system identifies x^0 with x, so as to prevent an ugly profusion of 0 subscripts. No such identification is made for a subscripted variable such as y^2, however. Such a variable would appear in the deduction as y^2^0. When resolving a clause with a constraint, variables in the clause are given a subscript one greater than the largest subscript used in deducing the constraint. No new subscript is introduced when resolving with a datum, nor by the special rules for ==, AND, OR and COND, which introduce no variables.

Variables in answers require somewhat more discussion.

If a variable from the query appears in an answer it appears in its original form, without the 0 subscript added at the start of the deduction. If a variable from a clause appears in an answer the treatment depends on the nature of the query. If it is a *primary* query, that is, one invoked from LISP, the variable simply appears with the subscript given in the deduction. If it is a *subsidiary* query, that is, one invoked recursively within some larger deduction, the query must have resulted from the reduction of an expression whose variables were given a subscript i > 0, while in the subsidiary deduction the variable was given a subscript j > 0. Such a variable, say x, appears in an answer (to the subsidiary query) as x^j^i. Since subscripted variables cannot appear in the knowledge base, this prevents unintended identification of variables in almost all cases of practical interest. We should point out, however, that if one clause causes two subsidiary deductions, and the answers to both contain variables introduced in the course of these deductions, it is conceivable that the same variable might appear in answers to both queries. Even in this case, such variables must appear inside quotations, and can enter the deductive process only if they are "exposed" by means of the special construct (LOGIC ...).

At this point the whole subject may seem overwhelmingly complicated, but we remind the reader that the programmer can ordinarily ignore the matter completely, and that the implementation achieves these effects implicitly and quite economically. In particular, variable identifiers like x^3^2 are never created in the internal workings of deduction; they arise only when needed for "export" to LISP. A very common particular case of exportation to LISP is, of course, when the variables are sent to be printed as part of an output.

CHAPTER 10

MONITORING DEDUCTIONS

Provision has been made for the optional "viewing" of a deduction process as it is happening. Ideally such a facility would show the tree of constraints growing during the execution of the deduction cycle. This would, however, be somewhat extravagant of display space, and LOGIC has a more modest version of this idea.

10.1 THE MONITOR FACILITY

Execution of the command (MONITOR :ALL) enables the system to display each successive selected constraint during the deduction process.

If the selected (implicit) constraint is (Q E) the display shows the (explicit) constraint Q*{E}. In order to give the user time to reflect, the system pauses once each cycle, and resumes on receiving a suitable input (normally, a C). The predications comprising the query Q*{E} are displayed as they exist before any simplification is performed.

It should be noted that when viewing a developing deduction process in this way one may observe some discontinuity in the display. This is because the selection mechanism may not always choose a successor of the previously selected constraint, but rather "resume" some older constraint whose turn has arrived for some more "progress". Even when the genetic thread remains unbroken, there may be rather drastic changes in the constraint owing to the LISP-simplification step of the cycle. The user will soon become accustomed to the realities of the MONITOR display, however, and will find it an enlightening tool when sparingly used to slow down and observe the deductive action.

The command (MONITOR :OFF) disables the MONITOR facility.

One need not simply continue from the MONITOR pause. The commands one can give are as follows (the prompt is "Monitor: (C, X, K, E, or Q)"):

C - Continue searching
X - Explain deduction
K - Discard these constraints
E - Evaluate an expression

Q - Terminate search

Responding to the MONITOR's prompt with the blue HELP key causes a help message (similar to the five lines above) to be printed. Any other input is ignored and a new prompt is issued. X, E, and HELP-KEY leave the system in the MONITOR pause. Both X (Explain) and E (Evaluate) ask the user for more input. In the case of X one may enter explanation qualifiers to specify the mode of explanation (see the next chapter). After entering E, the system prompts for the expression to evaluate.

10.1.1 Controlling The MONITOR Facility

One may wish to monitor only selected steps in the deduction. To do so, one executes the command (MONITOR P1...Pn) for some predicates P1...Pn. Thereafter, the system will monitor just those cycles for which the selected constraint begins with a predication whose predicate is among P1...Pn. We say that the predicates specified have been "flagged" for monitoring. To monitor empty constraints (successes), flag the identifier NIL, with, perhaps, other predicates. One can flag additional predicates by executing a similar MONITOR command, or "unflag" certain predicates with a command (UNMONITOR P1...Pn). (UNMONITOR :ALL) unflags all currently flagged predicates.

The (MONITOR :OFF) and (MONITOR :ALL) commands operate independently of flagged predicates, and without changing the flags. The command (MONITOR :ON) re-establishes selective monitoring.

One may also wish to observe constraints after simplification as well as before. The command (MONITOR 2) causes the system to print the constraint after simplification, in addition to the normal display before simplification, provided that the constraint was altered in some way by simplification. If selective monitoring is in effect, the decision as to whether the cycle should be monitored at all is still based on the initial predicate of the selected constraint, before simplification. The command (MONITOR 1) restores the normal mode, printing the constraint before simplification only.

The numerals "1" and "2" can be included in MONITOR commands which flag predicates, in which case they have the same effect as when they stand alone. The key words :OFF , :ON and :ALL are not recognized in such commands, however, so the command (MONITOR :OFF Male) would flag the predicates :OFF and Male and enable selective monitoring.

10.2 THE PURR FACILITY

It is often desirable to be able to see in some direct way that the deduction process is taking place, without necessarily slowing it down to the extent that the MONITOR facility entails.

The command (PURR :ALL) enables just such a facility, the PURR facility.

The PURR facility consists of a running display accompanying the deduction process. It involves the printing of a few single characters per cycle. No line feed is given after printing (except at the physical end of a line) so that the characters form a continuous string. The meaning of each character is as follows:

Character	Meaning
[	Start of a new query
- (hyphen)	Start of a new cycle
P	Selected constraint a success
U	Selected predication is NIL (false)
R	Resolvents of selected constraint obtained
X	Selected constraint failed for lack of resolvents
C	A continuation popped up
L	Selected constraint failed due to window limit
]	Completion of a query

The PURR facility is disabled by the command (PURR :OFF).

Thus with the PURR facility on the following transaction would occur:

```
(ALL (x y) (Age x 1920 y))[-R-R-R-P-R-P]
((Turing 8.) (Herbrand 12.))
```

The "PURR string" shows that the deduction took six cycles, invoked four procedures and found two answer environments.

Note that if a query is invoked within the processing of another query the PURR string will contain nested bracket pairs. As with the monitor facility, control is based on the initial predicate of the selected constraint, and predicates are flagged for purring with commands of the form (PURR P1...Pn), unflagged with commands of the form (UNPURR P1...Pn). Empty constraints (successes) are selected by flagging NIL, as with MONITOR. The key words :OFF, :ON and :ALL are used exactly as with MONITOR. Numerals are allowed in PURR commands, but have no effect.

One may nest calls of PURR and MONITOR, as in (PURR (MONITOR :ALL)), which enables both PURRing and MONITORing on all cycles. The same is true of UNPURR and UNMONITOR.

CHAPTER 11

EXPLAINING DEDUCTIONS

Once a deduction has been completed and its answer list obtained, one may call for an explanation of the reasoning by which some or all of the answers were deduced.

For instance, the following transaction consists of first constructing the answer list for the query (ALL (x y) (Age x 1920 y)) and then requesting an explanation for the second item.

```
(ALL (x y) (Age x 1920 y))
((Turing 8.) (Herbrand 12.))

(EXPLAIN 2)

To show·
((Age x 1920. y))

it is enough, by
(AGE-RULE (Age x 1920. y) <-
          (Born x [] [] birth-year1) &
          (== y 1920. birth-year1)))

to show:
((Born x [] [] birth-year1) (== y 1920. birth-year1)))

then it is enough, by
(HERBRAND1 (Born Herbrand 12. February 1908.))

to show:
((== y 12.))

then it is enough, by
(REFLEXIVE-LAW (== Reflexive Law))

to show:
NIL
DONE
```

The (EXPLAIN 2) command causes an explanation of the answer (Herbrand 12.) to be printed. The successive constraints leading to the answer are exhibited, and the clause activated to cause each transition is shown. The activated clause is shown with respect to the environment part of the resulting constraint (i.e. after the activation has extended the environment).

Various further inflections are provided with the EXPLAIN command. (EXPLAIN :ALL) provides explanations of all answers.

(EXPLAIN N1 ... Nk) provides explanations of the N1st, ..., Nk'th answers. (EXPLAIN) is the same as (EXPLAIN 1).

Explanations can be produced only when the history facility is enabled, which normally it is not. The history facility is enabled by (HISTORIES :ALL), disabled by (HISTORIES :OFF). Enabling the history facility can impose significant overhead on the system, particularly when the deduction tree must be searched to great depth.

The answers which one can have explained are those produced by the most recently completed invocation of ALL, ANY, THE or SETOF. If there are no such answers EXPLAIN will simply respond

```
Nothing to explain
DONE
```

An attempt to select a non-existent answer will be ignored, except that a note to that effect is typed.

11.1 ALTERNATIVE EXPLANATION MODES.

The EXPLAIN facility is considerably more flexible than indicated by the example just discussed, which illustrates only the normal mode of explanation. One can obtain explanations in a variety of styles. The variations are specified by typing qualifiers in the command following the selection of the answers to be explained. To illustrate, the command (EXPLAIN 2 :NAMES :FINAL) would print a similar sort of explanation, except that only the names of the clauses would be printed, and the constraints would all be recursively realized in the solution environment.

11.1.1 Specifying Items To Be Included.

Besides constraints and clauses, one may also instruct the system to print answer templates at each stage of the explanation, instantiated and simplified. One may also print names of clauses rather than printing clauses in full.

When names of clauses are to be printed the system will construct names for clauses for which the user has not specified names.

These "manufactured" names have the form (Pred :RULE k) or (Pred :DATUM k), following the conventions discussed in Chapter 6. User-supplied names are usually taken just as specified, but one can request "long" names, in which case the name given by the user is combined with the principal predicate symbol to form a list "(Pred Name)". Manufactured names are always in the long format.

The qualifiers which control all this are the following:

:ASSERTIONS	Print clauses in full	[Default]
:NAMES	Print names of clauses	
:UNNAMED	Print clauses which lack user-supplied names, print names where available	
:LONG	Print all names in long format	
:SHORT	Print user-supplied names in short format	[Default]
:CONSTRAINTS	Print constraints	[Default]
:NOCONSTRAINTS	Omit constraints	
:ANSWERS	Print answer templates	
:NOANSWERS	Omit answer templates	[Default]
:CONTINUATIONS	Print continuations with constraints	
:NOCONTINUATIONS	Omit continuations	[Default]

If :NOCONSTRAINTS is specified the format of the explanation is adjusted accordingly. If :NOCONSTRAINTS, :NOANSWERS and :NAMES are all specified the explanation is simply a list of the names of the clauses used, with no ornamentation. The default selection between :CONTINUATIONS and :NOCONTINUATIONS can be changed by (CONTINUATIONS :ON) or (CONTINUATIONS :OFF).

If *ASRNNUMBERS is non-NIL then anonymous clauses printed in explanations will include numbers, as explained earlier in Chapter 6.

11.1.2 Specifying Environments To Be Used.

We remarked earlier that the normal explanation shows each step of the derivation in the environment current at that step. One can, however, specify other choices as follows:

:INITIAL	Use initial (empty) environment	
:CURRENT	Use current environment	[Default]
:FINAL	Use final (solution) environment	

When the :INITIAL environment is specified constraints are shown in the current environment, as nothing earlier makes any sense, while clauses are shown in the form in which they appear in the knowledge base. Note that the :ANSWERS option is useful only in conjunction with :CURRENT, though other combinations are allowed.

Anything other than a qualifier appearing in the command will be ignored, with a warning message to that effect typed to the user.

11.2 LIMITING EXPLANATIONS

The full explanation of an answer, as normally produced by LOGLISP, can be quite lengthy, and one might wish to limit the explanation by omitting certain uninteresting steps. If one sets (HISTORIES :ON) then histories recorded by the system will include only those deduction steps which use clauses from procedures whose predicate symbol has the attribute :HIST, as might be declared by the command (PROCEDURE Pred :HIST) [see Chapter 5].

The following example shows the effect of including only steps using the procedures Age and = in the deduction of Herbrand's age in 1920.

```
(ADD-DECLARATION :HIST Age =)
:HIST

(HISTORIES :ON)
ON

(ALL (x y) (Age x 1920 y))
((Herbrand 12.) (Turing 8.))

(EXPLAIN 2)

To show:
((Age x 1920. y))

it is enough, by
(AGE-RULE (Age Herbrand 1920. y) <-
          (Born Herbrand [] [] 1908.) _&
          (== y (- 1920. 1908.)))

to show:
((== y 12.))

then it is enough, by
(REFLEXIVE-LAW (== Reflexive Law))
```

```
to show:
NIL
DONE
```

One observes that the omitted steps are not entirely ignored, since the bindings these introduce may influence the appearance of the steps which are retained in the explanation.

11.3 OBTAINING EXPLANATIONS IN LISP.

The system contains a number of functions which allow the LISP programmer to get at the basic material of the explanations. The programmer can then format explanatory material in whatever way he finds convenient. The first argument to each of these functions is an "answer number", which is the number of the answer to be explained, just as might be typed to EXPLAIN. The effect on these functions of predicates with the :HIST attribute is analogous to the effect on EXPLAIN.

(EXPLNAMES ANSNMB)

returns a list of the names, in long format, of the clauses used to derive the answer, in the order used.

(EXPLASSERTIONS ANSNMB ENV)

returns a list of the clauses used to derive the answer, in the order used. Here ENV should be one of the atoms :INITIAL, :CURRENT, :FINAL, to specify the environment in which the clauses will be shown. Each clause is represented by a list

(Pred Datum/Rule Name/Number Head T1 ... T1)

where "Pred" is the principal predicate symbol, "Datum/Rule" is either the identifier :DATUM or the identifier :RULE, according to the classification of the clause, "Name/Number" is the user-supplied name or system-manufactured number, and the remaining entries are the predications of the clause.

(EXPLCONSTRAINTS ANSNMB ENV CONTNS)

returns a list of the constraints arising in the derivation, beginning with the original query and ending with NIL. Here ENV specifies the environment as before, except that :INITIAL is treated the same as :CURRENT. CONTNS should be T if continuations are desired, NIL otherwise. The entries of the list returned by EXPLCONSTRAINTS are themselves lists of some complexity. If the constraint in question has no continuation, the corresponding entry has the form:

```
((q1 ... qN))
```

where qi is a predication. If the constraint has a continuation, but CONTNS is NIL, the entry will have the form

```
((q1 ... qN) CONTINUED)
```

while if the constraint has a continuation and CONTNS is T, the entry has the form

```
((q1 ... qN) (p1 ... pM) ...)
```

where pi is a predication of the continuation, which may itself be followed by another continuation, and so on.

(EXPLTEMPLATES ANSNMB)

returns a list of answer templates shown in the successive :CURRENT environments, beginning with the original template and ending with the actual answer.

All of these functions follow a common convention regarding exceptions. If the answer number specified does not correspond to an existing derivation the result is the atom :NO-EXPLANATION. If the most recent search was performed with the history facility disabled the result is NIL.

CHAPTER 12

INTERACTING WITH LOGLISP

In the present chapter we discuss the mechanics of running LOGLISP, obtaining information, controlling the operating modes and default settings, and some points dealing with errors.

12.1 RUNNING LOGLISP

Before running LOGLISP, LOGLISP must be loaded. To load LOGLISP, simply type at a lisp listener the following:

```
(LOAD "<fs>:<dir>;LOAD")
```

The Lisp Listener will respond with:

```
Loading <fs>: <dir>; LOAD.LISP > into package USER
Loading <fs>: <dir>; PACKAG.LISP > into package USER
Loading <fs>: <dir>; GLOBAL.LISP > into package USER

#<FS:LM-PATHNAME "<fs>: <dir>; LOAD.LISP#>">
```

where <fs> and <dir> are the file system and directory, respectively, on which the LOGLISP system lives. To complete the loading process, type:

```
(LOAD-LOGLISP)
```

The Lisp Listener will respond with:

```
Loading <fs>: <dir>; SUPORT.LISP > into package LOGLISP
Loading <fs>: <dir>; USER-INTERFACE.LISP > into package LOGLISP
Loading <fs>: <dir>; ENVIRX.LISP > into package LOGLISP
Loading <fs>: <dir>; UNIFCN.LISP > into package LOGLISP
Loading <fs>: <dir>; SUBSCR.LISP > into package LOGLISP
Loading <fs>: <dir>; SHOWNG.LISP > into package LOGLISP
Loading <fs>: <dir>; RDUCTN.LISP > into package LOGLISP
Loading <fs>: <dir>; PROCBS.LISP > into package LOGLISP
Loading <fs>: <dir>; RESLTN.LISP > into package LOGLISP
Loading <fs>: <dir>; HEAPMX.LISP > into package LOGLISP
Loading <fs>: <dir>; UNDFPR.LISP > into package LOGLISP
Loading <fs>: <dir>; SEARCH.LISP > into package LOGLISP
Loading <fs>: <dir>; SCHIFC.LISP > into package LOGLISP
Loading <fs>: <dir>; PRNTNG.LISP > into package LOGLISP
Loading <fs>: <dir>; EXPLNG.LISP > into package LOGLISP
Loading <fs>: <dir>; EDITNG.LISP > into package LOGLISP
Loading <fs>: <dir>; SAVRST.LISP > into package LOGLISP
Loading <fs>: <dir>; CONTRL.LISP > into package LOGLISP
Loading <fs>: <dir>; SYSINT.LISP > into package LOGLISP
Loading <fs>: <dir>; MISCEL.LISP > into package LOGLISP
Loading <fs>: <dir>; MENU.LISP > into package LOGLISP

LogLisp, version V3M1X4-Z
Copyright 1984, Syracuse University
<current time and date>
***
```

LOGLISP is now ready to use.

12.2 INITIALIZATION

The system starts out with an empty knowledge base. One may re-initialize the LOGIC part of the system at any time by invoking the function START.

```
(START)
```

leaves an empty knowledge base and resets the operating mode controls and system defaults to their standard values. LISP function definitions, file descriptions, and identifier values are not changed, except for those values which are used in system control.

12.3 INFORMATION

When interacting at the top level of LOGLISP, i.e. typing at a Lisp Listener, brief instructions may be obtained by typing HELP. A complete list of the functions which constitute the user interface to LOGIC, along with some other useful functions, is typed by the command (HELP), which may also be typed at the top level of LISP. In the MONITOR pause, in FACTS, and other times when LOGLISP is asking the user for input, help may be obtained by striking the blue HELP key.

Abbreviated instructions for using any of the LOGIC interface functions (ASSERT, THE, ALL, ANY, etc.) can be obtained by invoking the command (DOCUMENTATION 'fn), where "fn" is the name of the function in question.

12.4 CONTROL

The earlier chapters of this report mention a number of functions used to control various operating modes, as well as several defaults used by the system. In this section we shall summarize the control functions and explain the treatment of defaults in somewhat greater detail.

12.4.1 Control Functions

With the exception of PURR and MONITOR (see Chapter 10), all of the control functions take one argument, which should be :ON or :OFF, and return the argument after altering the system state appropriately. These function calls may be nested. To illustrate, the command (HISTORIES (CONTINUATIONS :ON)) enables both the recording of HISTORIES and the printing of CONTINUATIONS.

Several of these functions operate simply by setting the value of a LISP identifier, in which case NIL represents :OFF, while anything else represents :ON. (PURR, MONITOR, and HISTORIES use :ALL to represent the state selected by :ALL.) The identifiers so used may be changed directly by LISP programs, or accessed by them as may seem useful. The table which follows lists the names of the control functions, the initial settings, and, where applicable, the identifier set by the function.

Function	Initial Setting	Identifier
PURR	:OFF	*PURR
MONITOR	:OFF	*MONITOR
CONTINUATIONS	:OFF	*CONTINUATIONS
HISTORIES	:OFF	*HISTORIES
ASK	:ON	*ASK
AUTO-==	:ON	[None]
AUTO-AND	:ON	[None]
AUTO-OR	:ON	[None]
AUTO-COND	:ON	[None]

Initial settings are reestablished by START. The facility controlled by ASK is described below in the discussion of errors.

12.4.2 Defaults

Both in specifying deduction windows and in requesting explanations the user normally relies on many defaults. These are not, in fact, determined rigidly by the sytem, but may be adjusted by the user. The standard default settings are, however, restored by (START).

12.4.2.1 Deduction Window And Search Defaults - The defaults for deduction windows and search parameters are the values of the LISP identifiers listed below, along with their initial values.

Identifier	Initial Value
*TREESIZE	:INF
*NODESIZE	:INF
*ASSERTIONS	:INF
*RULES	1500
*DATA	:INF
*CSTEP	4
*ALLSTEP	64

Each of these gives the default value for the corresponding window limit. The implementation constraint on the number of rules in a single deduction will be rigorously enforced, even if *RULES is made larger than this limit.

The values which one may assign to these identifiers are the atom :INF or any non-negative integer.

12.4.2.2 EXPLAIN Defaults - The default qualifiers for EXPLAIN are similarly controlled by a collection of LISP identifiers. The table below shows the identifiers, the set of values each is allowed to take, and the initial value.

Identifier	Value Set	Initial Value
*EXPLASSERTIONS	{:ALL :SOME NIL}	:ALL
*CONSTRAINTS	{T NIL}	T
*LONGNAMES	{T NIL}	NIL
*ANSWERS	{T NIL}	NIL
*CONTINUATIONS	{T NIL}	NIL
*ENVIRONMENT	{:FINAL :CURRENT :INITIAL }	:CURRENT

Note that *CONTINUATIONS is controlled by the function CONTINUATIONS, and affects the monitoring facility as well as EXPLAIN.

12.5 ERRORS

Errors can arise either in LOGIC or in LISP.

12.5.1 LISP Errors

Errors detected by LISP will result in entry to the LISP error handler in the usual way. If the error arose during reduction a backtrace will show none of the workings of the reduction machinery, which is probably the best course the system could take.

All of the LISP facilities for recovery and analysis are available.

Note that misspelled function names in LOGIC terms will not lead to undefined function errors, simply to expressions which are not evaluable.

Although the LISP recovery facilities are available, one should not attempt to edit assertions during a break.

12.5.2 LOGIC Errors

Earlier chapters explained how syntax errors are handled by ASSERT and FACTS. There is one other type of error which can be detected by LOGIC -- the "undefined predicate" error.

A predicate is considered to be undefined if it has neither a LISP definition (as a function) nor a LOGIC definition (as a procedure of one or more assertions). If such a predicate is encountered during a search, and if the ASK facility is enabled (as it is initially), the system will ask the user for instructions, after first printing a message specifying the undefined predicate.

The prompt for instructions is:

```
What's next? (C, F, S, E, P, or Q)
```

The following help text will be displayed if the user strikes the blue HELP key:

```
Respond with:
     C to continue searching
     F to execute FACTS
     S to correct spelling
     E to evaluate an expression and print the result
     P to print the current state
```

Q to abandon the search

Anything other than the inputs specified causes the system to remain in the ASK state. If the user does anything which might conceivably alter matters, the system will try again to simplify and obtain resolvents.

The automatic spelling correction attempts to find a predicate (defined by LOGIC) which closely matches the undefined predicate. If successful it informs the user of the chosen predicate, if not successful it informs the user of that fact. Spelling corrections are accomplished with RPLACA, so the effect may reach beyond the immediate situation. When the undefined predicate occurs as an instance of some variable, spelling corrections are probably unwise, and the user is warned of such circumstances. Afterwards it may help to run with (HISTORIES :OFF).

12.5.3 LOGLISP Utilities

Some of the LOGLISP system utility functions may be of use to programmers. The names of these functions are not reserved.

(VARIABLE e) [FUNCTION]

returns T if e is a LOGIC variable, NIL otherwise. The closely related form (Variable "e") may conveniently be used in assertions to distinguish variables from other expressions (see Chapter 4).

(Version) [FUNCTION]

prints a message identifying the version of LOGLISP in use.

(HELP) [FUNCTION]

prints a classified list of logic system functions.

(RTIMER "expr") [MACRO]

evaluates "expr", prints timing information about the evaluation, and returns the value thus obtained. RTIMER prints the number of seconds spent evaluating expr, the number of resolvents computed, the rate at which they were computed (in LIPS "logical inferences per second"), the number of simplifications performed, and the combined rate at which both resolvents and simplifications were performed ((resolvents + simplifications)/seconds). By a simplification we mean the evaluation of one predication by LOGIC. Subsidiary evaluations performed in the course of reduction are not counted as simplifications.

REFERENCES

[Boyer-Moore 1972]

Boyer, R. S., Moore, J. S. — The Sharing of Structure in Theorem Proving Programs. Machine Intelligence 7, Edinburgh University Press, 1972.

[Bruynooghe 1976]

Bruynooghe, M. — An Interpreter for Predicate Logic Programs, Part I. Report CW 10, Applied Mathematics and Programming Division, Katholieke Universiteit, Leuven, Belgium.

[Clark 1979]

Clark, K. L., McCabe, F. — Programmers' Guide to IC-PROLOG. CCD Report 79/7, Imperial College, University of London.

[Colmerauer 1973]

Colmerauer, A., Kanoui, H., Pasero, R., Roussel, P. — Un Systeme de Communication Homme-Machine en Francais. Rapport, Groupe Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, France, 1973.

[Floyd 1964]

Floyd, R. W. — Algorithm 245, TREESORT [M1]. Communications of the Association for Computing Machinery 7 (1964), p. 701.

[Hill 1974]

Hill, R. — LUSH Resolution and Its Completeness. DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

[Kowalski 1974]

Kowalski, R. A., Predicate Logic as Programming Language. Proceedings IFIP Congress, 1974.

[Kowalski 1979]

Kowalski, R. A., Logic for Problem Solving. Elsevier North Holland, 1979.

[Meehan 1979]

Meehan, J. A., The New UCI LISP Manual. Lawrence Erlbaum Associates, 1979

[Moon Stallman and Weinreb 1983]

Weinreb, D., Moon, D., Stallman, R. Lisp Machine Manual. Fifth Edition, System Version 92, January 1983.

[Moon and Weinreb 1980]

Weinreb, D., Moon, D. Lisp Machine Manual. Second Edition, March 1980.

[Roberts 1977]

Roberts, G., An Implementation of PROLOG. M.Sc. Thesis, University of Waterloo, 1977.

[Robinson 1965]

Robinson, J. A., A Machine-Oriented Logic Based on the Resolution Principle. Journal of the Association for Computing Machinery 12, 1965, pp. 23-41.

[Robinson 1979]

Robinson, J. A., Logic: Form and Function. Edinburgh University Press and Elsevier North Holland, 1979.

[Robinson-Sibert 1980]

Robinson, J. A., Sibert, E. E. Logic Programming in LISP. Technical Report, School of Computer and Information Science, Syracuse University, November 1980.

[Robinson-Sibert 1981]

Robinson, J. A., Sibert, E. E. — LOGLISP Implementation Notes. Technical Report, School of Computer and Information Science, Syracuse University, December 1981.

[Robinson-Sibert 1984]

Robinson, J. A., Sibert, E. E. — LOGLISP Implementation Notes. Technical Report, Logic Programming Research Group, Syracuse University, February, 1984.

[Roussel 1975]

Roussel, P., — PROLOG: Manuel de Reference et d'Utilisation. Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, 1975.

[van Emden 1977]

van Emden, M. H., — Programming in Resolution Logic. Machine Intelligence 8, 1977, pp. 266-299.

[Warren 1977]

Warren, D.H.D., Periera, L. M., Pereira, F. — PROLOG - The Language and Its Implementation Compared With LISP. Proceedings of a symposium on AI and Programming Languages, SIGPLAN Notices, Vol. 12, No. 8, and SIGART Newsletters 64, August 1977, pp. 109-115.

INDEX